The preservation of the environment.

Padget, Julian Alexander

Award date:
1984

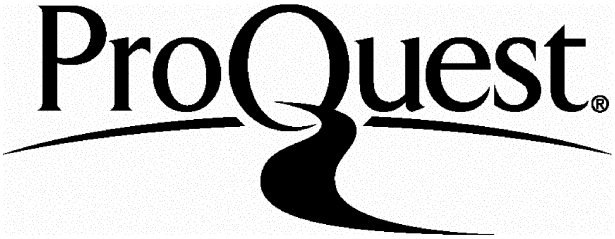
Awarding institution:
University of Bath

The Preservation of the Environment

submitted by Julian Alexander Padget
for the degree of Ph. D.
of the University of Bath

1984

J A Padget



Contents

Acknowledgements

Many people have aided me in various ways during the past three years and more. Consequently they have all had varying degrees of influence on the work presented in this thesis. The conversation, insight, brow-beating, understanding, coffee and encouragement of the following is gratefully acknowledged:

KC

JPff AMff MM

JKS JHD ACN RS IH

BWS JBH PJWI WG DHS

APPH CACG IBF

JEÅ JBM

Summary

A new binding model which permits the capture of both dynamically and statically scoped environments is presented. The basis of the model is the technique of environment labelling.

The practical side of this work has been carried out in the context of the programming language LISP. Various timing tests have been run and the new scheme compares favourably with the existing environment models and variations on them. To date the two main approaches to environment representation, shallow and deep binding, have suffered from each having just one strength. Deep binding is efficient at context switching whilst shallow binding is good at variable access. This new model attempts to provide both these facilities at comparable efficiency.

Consequently, the user is now given direct access to the environment from the programming language allowing the construction of continuations which radically enhances the expressive power available. Many interesting algorithms which were previously impossible because environments were allocated on a stack are now opened to investigation. The thesis finishes with a discussion of some suitable topics where this approach may be applied.

Introduction

The long term aim of the work presented in this thesis is to provide a foundation for a reconsideration of the semantics of LISP [McCarthy *et al.* 60]. The results of this research (environment labelling and a new binding model) are general, but the majority of the practical effort has been directed toward extending a particular implementation of LISP [Fitch & Norman 77].

During the evolution of LISP one major feature was sacrificed in the cause of efficiency. That feature was the FUNARG. The main purpose of this work has been to reinstate this facility as part of a wider intention to restore the original semantics of LISP. The next logical stage is to refine and extend the semantics by removing inconsistencies and making generalisations such as the addition of local variables in the interpreter and general scoping of data structures. Local variables have two advantages: compilation (where this means a semantics preserving program transformation process) may make those variables into frame locations, which is faster than named lookup, and secondly, lexical closures are often sufficient for many applications (e.g. simple generators). General scoping provides an ability to scope arbitrary objects in conjunction with access and update functions which know how to interpret the scope information. This is useful both for security and for context sensitive programs (e.g. AI knowledge bases).

The thesis is that dynamic scoping is too important and too useful to discard completely as has been suggested [Sussman & Steele 75]. Part of the reason for the movement away from the dynamic scoping model including functional values is the high toll in efficency paid either all the

time or at an extortionate rate when the facility is actually used. These two points highlight the problem: the difficulty in providing a fast implementation. This should not be taken as reason enough to adopt purely static scoping, rather a challenge to find a better solution. It is obvious that the provision of multiple environments must entail some cost over a model that does not include an environment component (e.g. shallow binding as generally implemented). That cost is because the binding interrogation function must look up an identifier with respect to an environment rather than just returning the contents of a single location related to the identifier.

The approach chosen determines where the cost will fall. There are three ways of modeling multiple environments:

(i) interrogate an identifier with respect to an environment

(ii) look up an environment with respect to an identifier

(iii) always look in the same place for the value of an identifier, but also record the differences between environments so that a previous one may be reinstated.

The first and third of these schemes are well known as deep binding and shallow binding respectively. This thesis develops and describes the second. The first method allows for fast context switch at the expense of variable interrogation. The third advances the opposite philosophy, where context switch is very slow but binding lookup is very fast. The second is an attempt at a unification of the two whilst tending to their respective *beneficial* extremes. Context switch is the same cost as for

the first model. Variable lookup is harder to classify being, in theory, unbounded in the same sense as deep binding but, in practice, the cost is always small (indeed bounded) for programs with stack-like behaviour and varies depending on the structure of the problem for programs which context switch frequently.

The basis of the implementation of the new model is the technique of environment labelling. This appears to be a very powerful handle on an environment and for discovering the relationships between environments. The scheme provides an easy method for handling both static and dynamic scoping of identifiers. Indeed it is generally applicable and may be used to scope within arbitrary data structures such as trees, property lists and vectors.

Chapter 0 contains a review of the work to date in this area: how the problem arose and was compounded and recent attempts to solve it. Chapter 1 introduces the key idea which lead to all of the work contained herein and concludes with a detailed schema for the rest of the thesis. Chapters 2-4 discuss implementation matters. Chapter 5 contains cost breakdowns and timings for the new model and its rivals. Chapter 6 is a defence of dynamic scoping and a discussion of the wider implications of labelling. Chapter 7 considers the ramifications for garbage collection. Finally, Chapter 8 goes on short excursions into various fields investigating how continuations can be used.

CHAPTER 0

History and Overview

Some history

The origin of the problem of implementing dynamically scoped closures dates back some twenty years. For this reason it is pertinent to give some history of how the problem arose, the consequences of decisions taken in those early days, and what attempts have been made in the intervening period to resolve the difficulty. The distinction between 'static' and 'dynamic' is an important one, in some respects the crux of the problem, and is discussed with respect to the alternative, lexical scoping, in greater depth later in this chapter.

The practical side of the work presented in this thesis is given in the context of the language LISP. It should not be construed as being restricted to LISP; the ideas are general and applicable to any language. What is more important is that they are developed and expressed in abstract concepts with strong mathematical foundations: formal semantics (Chapter 1) and graph theory (Chapter 2).

The purpose of this chapter is to provide some sort of context for the work which will be presented in the following chapters. Because this work is intimately bound up with the history of LISP, it is also necessary to relate some of the watersheds in its development, and to explain some of the rationale behind the various decisions. Events of importance are: deep binding; the recognition of the FUNARG problem; shallow binding; the provision of multiple control environments; and attempts to recover the functionality lost when accepting shallow binding.

It is appropriate at this stage to introduce some of the terms which will be used in the following discussion. A *function* is taken to mean an expression which may be applied to arguments. This means it must either be a form whose first element is LAMBDA or it must be a member of the distinguished set of datatypes known as code-pointers which are primitive values created by the compilation of forms of the first category. It does not include any environmental information: that is only necessary in a statically scoped language where functions have to be closed at the of point declaration. The main concern of this thesis is with the object referred to above as a dynamically scoped closure. That is probably the most accurate phrase to use, but it is long and tedious in repetition, and on occasions, this may be truncated to *closure*. Synonymous terms are *environment* or *context*, meaning the set of values bound to all the free and local variables visible at a particular instant during the evaluation of a program. An environment is of little use in isolation, but must be coupled with an *expression*. Similarly an expression has no concrete meaning when regarded alone, only a *meta*-meaning in that it describes an abstract expression. To recover a meaning, it must be evaluated with respect to an environment. Such a pairing may also be referred to by several names, for example FUNARG, which is an historical relic inherited from the language LISP, or *continuation*, which is a more recent invention of the semanticists. Continuations themselves come in two forms, the command continuation and the expression continuation. The difference between the two will be explained later (see Chapter 1 and Chapter 6), but we shall be concerned in the main with the latter.

The problem of the dynamically scoped closure arose during the first implemention of LISP [McCarthy 60] at the Massachussetts Institute of

Technology (MIT) in 1958. Some detailed history is found in [McCarthy 77]. LISP is one of the only three languages (the others being APL and SNOBOL) in widespread use which supports dynamic scoping of variables. In brief, dynamic scoping says that the extent of a variable includes all the functions called from the block in which the variable is bound. Consequently, the scope of a variable is controlled by the name (identifier) being rebound somewhere in the sequence of function calls. By contrast, under lexical scoping, the extent of a variable is limited to the block in which it is bound and any other blocks which that block encloses *textually*. Hence the scope of a variable is limited by being rebound somewhere inside the block nest. Consider the diagram of a set of program contours shown in Figure 0.0, and then the interpretations beneath.

There is clearly only one possible mapping between the contour diagram and a piece of code given lexical scoping. There are many mappings given dynamic scoping including the lexical version, which suggests that dynamic scoping in some sense subsumes lexical scoping. This multiplicity of mappings is the heart of the difficulty in implementing a dynamically scoped language. The implicit restriction of lexical scoping allows determination of the location of the value of a non-local identifier in the binding stack to be done at compilation time. That is not possible with dynamic scoping, since there can be no *a priori* stack allocation for free variables. A method is needed to associate a name with a value (its binding). A simple scheme for this will now be described.

The first binding scheme

The first technique developed is generally known as deep binding. It was originally conceived as a stack oriented method: the demand for

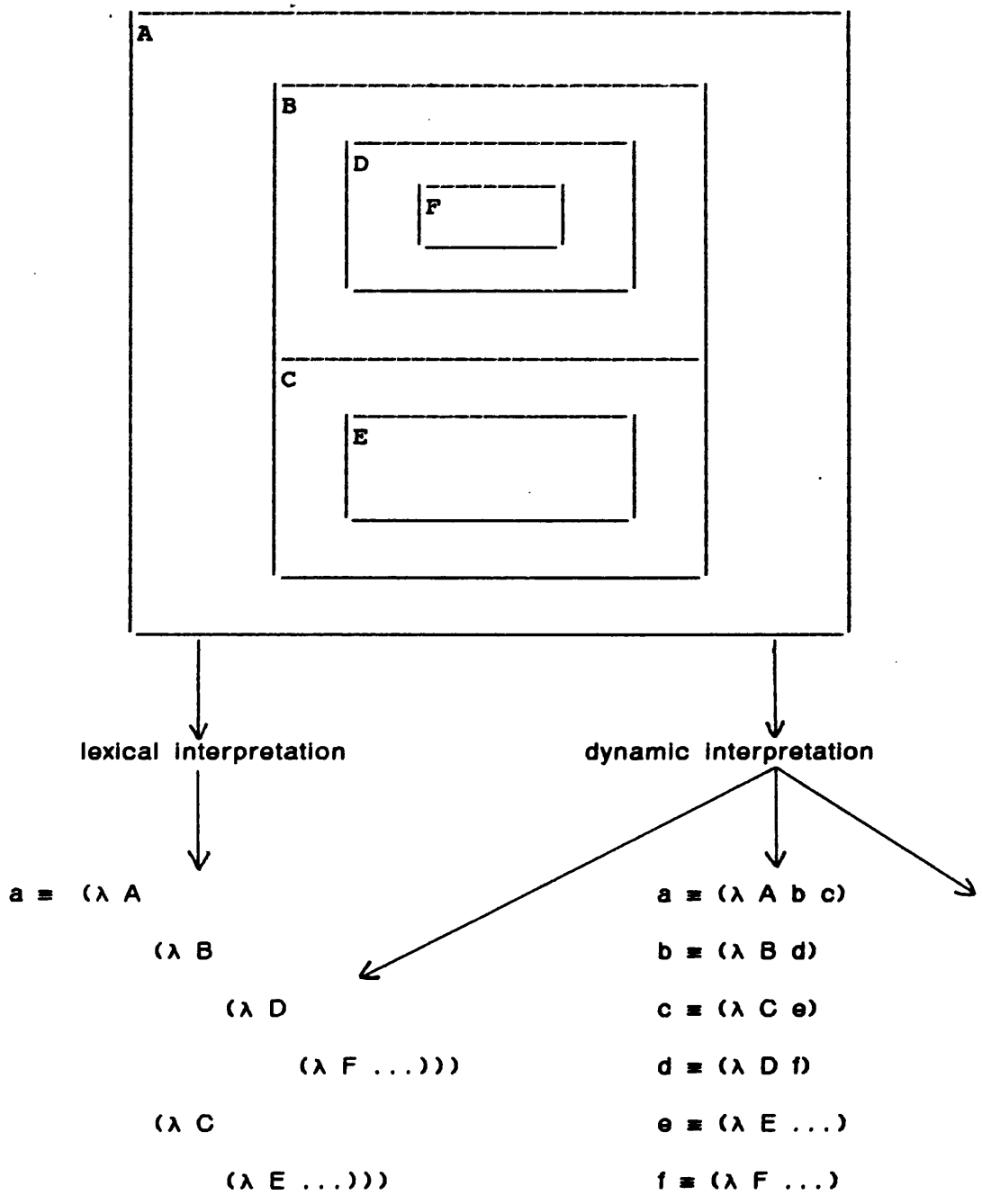

A
B
D
F
C
E
lexical interpretation
dynamic interpretation
a ≡ (λ A
(λ B
(λ D
(λ F ...)))
(λ C
(λ E ...)))
a ≡ (λ A b c)
b ≡ (λ B d)
c ≡ (λ C e)
d ≡ (λ D f)
e ≡ (λ E ...)
f ≡ (λ F ...)

Figure 0.0

FUNARG changed that. When starting the evaluation of a lambda expression, the names given in the formal parameter list are paired with the values just computed in the actual parameter list and pushed on to the binding stack (also known historically as the a-stack for 'association stack'). When the evaluation of the lambda expression is complete the set of bindings which were created for it are popped off the a-stack (see Figure 0.1). To find the value of a variable, the binding stack is searched from top to bottom, using the name as the search key. The first pairing of that name encountered gives its current value. This method was employed in the first implementation of LISP (LISP 1.0 on the IBM 704). The result was a first order functional language with dynamic scoping.

Computational models

LISP was conceived as a mechanical and intentionally impure evaluator of lambda calculus expressions [Church 40]. It did not make use of the reduction schemes described by Church and later by Rosser [Rosser 35] despite their proven mathematical capabilities. In fact reference to the description of the binding process above reveals the use of applicative order (left to right) evaluation. Nor did the implementors take great heed of the normalisation or standardisation theorem [Barendregt 81]. It had been shown that λ-calculus has the diamond property: to reduce an expression M to an expression M' where more than one β reduction (substitution operation) is required, then those reductions may be done in any order (see Figure 0.2). One such order is known as applicative order, another is called normal order. However the Standardisation Theorem states that if a canonical form exists, then normal order evaluation will produce it. This cannot be guaranteed using applicative order. Reasons for these decisions, apart from

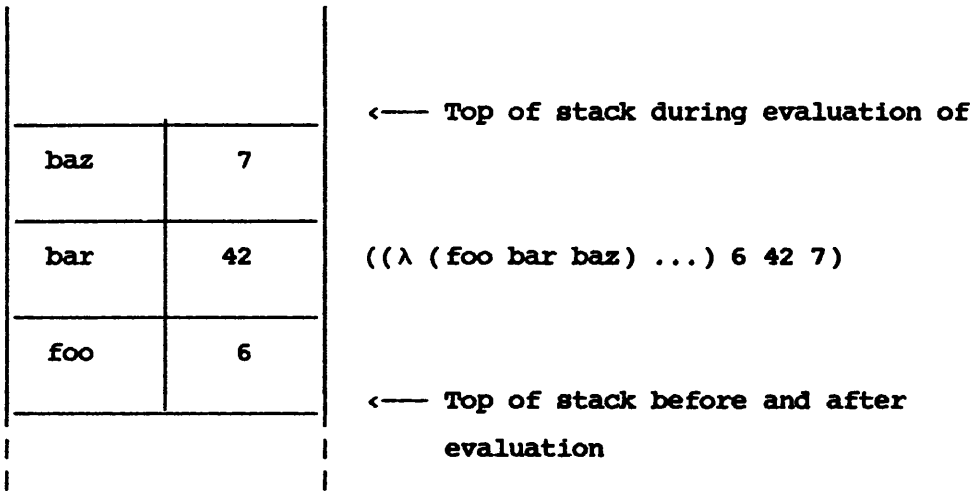


Figure 0.1

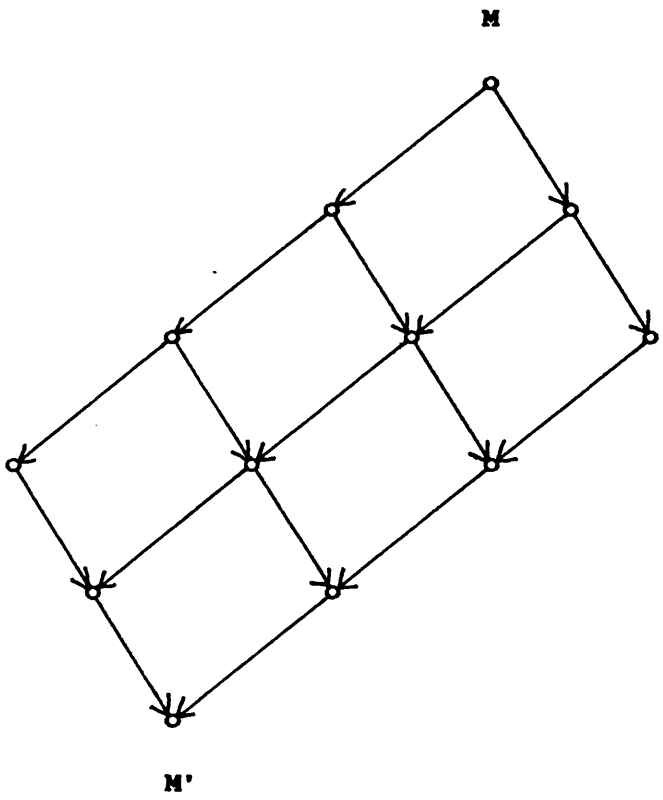


Figure 0.2

implementational expediency, are unclear.

An alternative approach is to discard variables entirely and use only functions to encode λ-expressions as suggested in [Turner 79a, 79b]. These papers describe language implementations based on functions called combinators [Schonfinkel 24] and the combinatory logic developed by Curry [Curry 30] [Curry & Feys 58]. This has both advantages and disadvantages, and is currently a topic of much research. The ease with which this scheme copes with high-order functions and multiple environments is particularly in its favour. In fact the problem is greatly simplified as combinator expressions contain no variables so there is no environmental information to maintain. Indeed, the evaluation mechanism is normal order by nature with attendant advantages. This loss of complexity must balanced against two things:

(i) a different execution strategy (graph reduction), which does not map very well onto existing machine architectures. This incompatibility is often said to be the limiting factor in the performance of such systems. Some specialist architectures show promise however [Clarke *et al.* 80], [Stoye *et al.* 84].

(ii) a purely functional programming language. Assignment cannot be supported by this methodology. It is yet to be shown whether this is a limitation or an advantage in practice.

McCarthy did consider the use of combinators, but concluded that they were too simple in themselves leading to large and complex expressions in practical applications. An example of the consequences of their simplicity is that it cannot be proven that two equivalent

combinator expressions are in fact equal e.g. both S K K and S S K form the identity function (known as I), but combinatory logic cannot show them to be equal, whereas the equivalent λ-expressions can be proven equal within the rules of the lambda calculus. More recent research [Turner 79a] [Burton 82a] has shown the second objection to be less valid by producing more compact expressions through the judicious addition of new combinators [Turner 79b]. The basic S and K combinators are sufficient to encode any expression but only using them leads to exponential growth. The new combinators bring the bound down to less than quadratic [Kennaway 82].

The development of LISP

LISP 1.0 was not intended to be the end of the story: two more stages were envisaged. The first is LISP 1.5 in which the destructive functions such as SET, RPLACA and RPLACD, and the FUNARG object were introduced. Finally LISP 2.0 [Abrahams 66] was intended to change the syntax from the heavily parenthesized form to something not dissimilar to ALGOL 60 notation. LISP 2.0 was never completed and remains something of a singularity in the evolution of the language. It suffered from the desire of the implementors to make it all things to all men (which has been the death knell of many an enhanced implementation). In addition LISP users were coding directly in the S-expression form rather than MLISP, as used to present the meta-circular interpreter in [McCarthy *et al.* 62], followed by manual translation as McCarthy had intended, which negated the greater part of the LISP 2.0 effort.

LISP was originally implemented to support research activity at MIT into recursion equations, recurrence relations and logic [McCarthy 73]. However it rapidly outgrew its original conception as a pedagogic toy.

Other researchers found that the style and extensible nature of LISP was well suited to such subjects as artifical intelligence, robotics and symbolic algebra, so demand turned it into a general programming language and caused the introduction of features beloved of other less soundly founded languages, such as assignment, sequencing, labels and jumps.

Another consequence of this unexpected popularity was a demand for greater efficiency. This had far reaching effects, especially clear now that they can be considered with hindsight. In the earliest LISP systems, functions (and user definitions) were global objects, but because of the problem of functional arguments, it was still necessary to carry out a full environment search when calling a function to check for a more recent binding. Obviously this was slow: there is a way round it, but it changes the semantics of the language. There is evidence [McCarthy *et al.* 62] to suggest that the implementors did not realise the consequences of their actions. An atom consisted of a a print name and a property list in these early days. The global value of an identifier was found under the 'APVAL (and latterly the 'VALUE) property and global function definitions under the 'EXPR, 'FEXPR, 'SUBR, 'FSUBR and 'MACRO properties. This meant a name could have several function bodies associated with it (or even the same body under different properties! - consider LIST and EVLIS). The one selected for application depended on the method employed to find the definition. Alternatively an identifier could be rebound and then applied by referencing the name in the function position of a form. The latest binding would be found in the course of the environmental search and used in the manner of an 'EXPR definition; this would indicate either a confusion as to the nature of values to be used as functions, or that the difficulty was never really

considered in depth. It also shows the folly of associating the function with the type ('EXPR, 'FEXPR, 'MACRO), as happens with the property list scheme, rather than the function with the type (e.g. LAMBDA implies evaluation of arguments and LAMBDAQ implies quotation of arguments).

The way to avoid the costly environmental lookup is to evaluate the function position of form in a different way from the rest of the expression, by looking directly at the global binding. Hence the use of a functional argument requires special treatment, so FUNCALL and APPLY* (depending on dialect) were introduced. For example

```
(de foo (bar baz) (bar baz))
```

may or may not work depending on the global value of bar, but it is unlikely to have the intended effect of applying the bound to bar function to baz. If bar is bound to the same value as the global function definition of bar, then the behaviour will be as expected. If bar and its global function value are different then there will either be an error or the function will not do what the programmer intended. To overcome this one must write

```
(de foo (bar baz) ({ funcall } bar baz))
                  { apply*  }
```

This 'optimisation' is accredited to D.G.Bobrow when he was implementing BBN LISP [Bobrow & Murphy 67] on the PDP1 in 1965.

The multiple environment problem

This became known in LISP circles as the FUNARG problem after a paper which purported to explain it [Weizenbaum 68]. The matter is also covered in some detail in [Moses 70]. The nub of the question is how to represent the environment (and manage the store which implements it)

so that a particular environment may be preserved for use later. In the discussion of FUNARGs, there are two types, those being passed as arguments (downward) and those being returned as values (upward). In many respects this distinction is unnecessary. An environment is an environment wherever it is, but because some LISP systems were not fully general in their implementation, they could only offer environments lower down the stack and not those that have already been exited.

The first widely used complete LISP with FUNARGs was INTERLISP [Teitelman *et al.* 72] which was developed from BBN LISP. Of course environments cannot be entirely stack-allocated to provide upward FUNARGs because the area beyond the top of stack might have to be retained after control has left it. One simple way to manage the storage of multiple environments is to allocate out of heap and let the garbarge collector recover function frames. This has several disadvantages:

(i) arguments must be stored in temporaries before being transferred to the frame

(ii) there is a fairly fast turnover of quite large amounts of store, hence garbage collection will be more frequent and more intrusive on a single processor non-incremental system

(iii) poor paging behaviour occurs becuase there is no contiguity of stack end

A closer analysis of the nature of the problem shows that some features can be capitalised upon for a more efficient system. An instance of this is the spaghetti stack model [Bobrow & Wegbreit 73] used

to implement both INTERLISP and YKTLISP [Blair 78]. In essence there is a separate heap for function frames, but because more is known about the behaviour of these objects it is easier to return redundant space explicitly to a free list. There is still occasionally a need to garbage collect and compact the space. The first INTERLISP system used deep binding, but the results of searches were cached into the function frame (in fact 'pushed' on to the top of the frame) for faster subsequent lookup. However still greater speed was sought, and the spaghetti stack system was very complex to implement. Both of these were causal factors in the development of a scheme now known as shallow binding.

Shallow binding

One way to minimize the cost of variable interrogation is to ensure that the current value can always be found in the same place. This location is often referred to as the value cell. This can be an extra slot in the structure used to represent an atom wherein the current value is kept, rather than on the property list. Some implementations have used the value cell to contain a pointer to the property list entry of the 'VALUE property. When an identifier is bound (or rebound), the old value (i.e. the current contents of the value cell) is saved into the function frame, and the new value is placed in the value cell. The unbinding process is the reverse of the above: the value in the function frame is put back into the value cell and the environment is as before.

There is a great price to be paid for this simplicity: context switches are now overwhelmingly expensive. The information held in the function frames only reflects the changes that have taken place in the environment. Nowhere is it possible to get a direct handle on the environment - it is distributed everywhere, all that is known is how it was

modified. This means that the process to restore another environment must descend the tree toward the root undoing the changes until a frame is encountered which is on the path to the root from the target environment. From there it ascends the branch carrying out the changes specified in the frame until the goal environment is reached. Obviously this could take a very long time, such that even if this feature is part of a system, it will only rarely be used. INTERLISP under TENEX on the DEC PDP-10 and DEC-20 supports this method [Teitelman 78]. The burning question is how to find the first common frame. In INTERLISP this is done by tracing from the target environment to the root, leaving a mark in each frame. Then the context switch operation descends the tree from the source environment until it finds a marked frame, at which it starts the ascent phase. It is significant that the next implementation of INTERLISP (known as INTERLISP-D) on the Dorado [Lampson & Pier 80] reverted to deep binding (albeit using the function cell trick discussed previously).

One further remark is that, although the cost of lookup on any item was now bounded, this did not mean in general that the function context would be treated in the same way as the rest of a form. From the adoption of the function cell until the advent of shallow binding many (and large) programs had been written which relied on that particular questionable feature - using a name which was bound at the global level to a function as a formal parameter identifier in LAMBDA and PROG expressions. This meant that the function itself was still available if needed, but that the same name used in an argument position had a different value, for example

```
        (de foo (list) (list list))
then  (foo '(a b)) → ((a b))
```

Consequently the cost of variable lookup (or function lookup - are they not the same?) becomes constant with shallow binding. To preserve the above feature, the function cell was invented (by analogy with the value cell), rather than let function values be kept in the value cell, which would have reinstated the original semantics.

LISP and AI application languages

Artificial intelligence research started realising that although it needed the speed of shallow bound systems, the simple control mechanism (first order functions) was a severe limitation, particularly with respect to searching trees of frames, and handling semantic networks. Something more poweful was required: INTERLISP had some of the flexibility but not the speed (at this time the DEC-10 was the major AI workhorse). This need was manifested in several research projects, notably PLANNER [Hewitt 72], CONNIVER [Sussman & McDermott 72a] and MDL [Galley & Pfister 75].

Both PLANNER and CONNIVER were very complex languages (such that PLANNER was never completely implemented). A common feature of these was the ability to backtrack: PLANNER took this too far in catering for the possibility of backtracking all the time, whilst disregarding the reason for the failure which could have been used to direct the search more profitably. PLANNER adopted the view that if there was no clear solution (either to the problem - or in the mind of the programmer), then given a lot of methods and heuristics, a backtracking control mechanism and a lot of computer time, it *might* arrive at a solution. This approach was justifiably criticised [Sussman & McDermott 72b] and helped crystallise the design of CONNIVER, which sought to remedy these deficiencies. The principle of CONNIVER was that the facilities of

PLANNER should be available to the programmer, but not automatically. The reasons are twofold: one, to improve efficiency and two, to encourage a deeper analysis of the problem by the programmer. The version of LISP described in this thesis subscribes to this view in that it provides a means of constructing complex control mechanisms, but their maintenance is the responsibility of the user.

MDL was conceived at MIT between 1970 and 1975 as the next generation of LISP. In particular, the designers had in mind the need for more exotic control mechanisms, such as processes, coroutines and generators. The environment was modelled using a modified form of deep binding. The optimisation was that each identifier had a value cell in a standard location (like shallow binding) and an environment pointer. On interrogating an identifier the value of the environment pointer is compared with the current environment, and if they are equal, the contents of the value cell are used. If the two pointers are not the same, the usual deep binding search method is employed, except that having found the appropriate binding, the value cell is updated, and the environment pointer changed to the current environment. The other major interesting feature of MDL is the provision of processes and multiple stacks, which formed the basis of the LISP machine LISP (LML) stackgroups mechanism. In fact LML [Weinreb & Moon 81] has quite a strong heritage in MDL in this and other rarely recognised ways.

The most fundamental difference between LML and MDL is the former's use of shallow binding. This raises the question of how to do context switches. The only possible way is the one mentioned earlier in the section on shallow binding, although that used a spaghetti stack rather than the stackgroup mechanism found in LML. The method was

described more formally in [Baker 78b], and will henceforth be referred to as *rerooting*.

Rerooting

The method set out in [Baker 78b] constructs the chain of bindings in the heap (although this could be adapted to fit into the spaghetti stack structure). The bindings are in the classical form of the association list (henceforth alist). It starts with the observation that all lookups are trivial if the current environment is the root of the evaluation tree (the correct binding is in the global value cell). If the binding process can can ensure that the newly created context becomes the root, then subsequent lookups will also be trivial.

The rerooting mechanism is developed inductively, that is first considering the necessary steps to 'move' the root from one node to an adjacent node, and then generalising this to moving n nodes. The current environment is regarded as the root of the evaluation tree; to change the root to an adjacent node, the link pointing from the target node to the current environment is reversed and then transversed, exchanging the contents of the value cell and the saved binding. Now, having arrived at the new root all the identifiers have the correct values (see Figure 0.3). Extension to rerooting to a non-adjacent node is straightforward: apply the rerooting algorithm as for one node, then repeat the process until the target node is reached (see Figure 0.4). Such a system is called continuously shallow bound. An interesting consequence of this method is that environmental extensions or context switches can be made from the root *without* needing to reroot the tree as long as the variable interrogation mechanism is changed thus. If the current environment is the root, use the value cell, otherwise use the

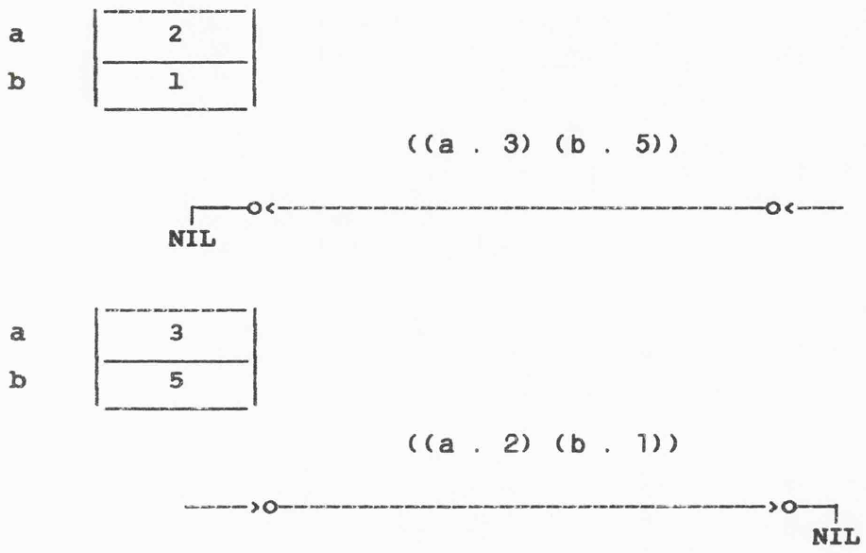


Figure 0.3

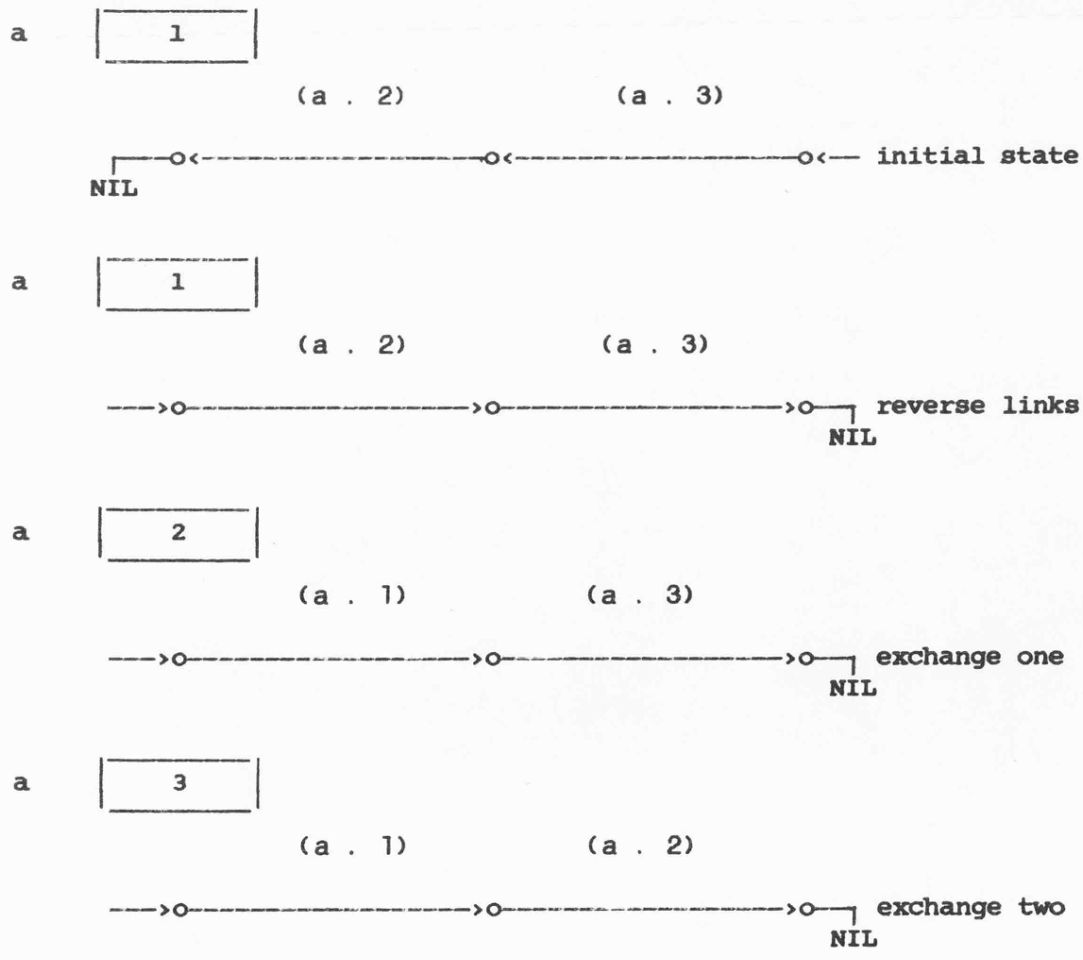


Figure 0.4

standard deep binding lookup. This is called casual rerooting. Details of a particular implementation of continuous rerooting are given in Chapter 5.

Chapter summary

This chapter has tried to set out the knowledge of binding models and their implications which existed when the work presented in the following chapters was started. The most important features to remember are the two major binding models developed over the last twenty years: deep binding and shallow binding, their strengths and their weaknesses. Attention is also drawn to methods of incorporating multiple access environments into shallow binding such as that found in INTERLISP-10 [Teitelman *et al.* 72] and latterly described in [Baker 78b] as *rerooting*.

CHAPTER 1

Semantics of the Binding Process

Motivation

Why the interest in an efficient closure mechanism? It has long been recognised that closures sometimes lead to either the only solution or at least a more elegant solution for certain complex programming problems. This is largely because the abstract notion of a closure is more closely related to the physical situation being modelled than any purely sequential form could be (for some examples of the situations referred to see Chapter 8).

It took some time to recognise that the closure (or the more fundamental continuation) actually had a strong theoretical basis, which goes, at least in part, toward explaining why it is so generally useful. In the late 1960s work on semantics was just beginning. In particular there was a group at Oxford University, comprised of amongst others Christopher Strachey, Chris Wadsworth and Dana Scott. At the latter end of the decade attention was turned to formalising the semantics of jumps and the use of labels. Few languages at that time supported the concept of the label as a first-class object, although CPL [Barron *et al.* 63] did consider the problem in more than passing detail. CPL permitted the programmer to go to a label regardless of whether it was presently in scope, the jumping process (re)instantiating the environment of the label as necessary. This idea was generalised in PAL [Evans 68], which was concieved purely for the purpose of teaching and experimenting with the semantics of complex control schemes.

The problem was how to express the semantics of two consecutive statements. What was difficult about this was the power and the danger of general jumping. It is not guaranteed that the second statement will be evaluated immediately after the first, since the first may jump somewhere. What was needed was a way of describing what the program would do next. Rather than just talking about the *one* following statement, we can talk about the following *group* of statements, that is the rest of the program coming after the current statement. Of course this conceptualisation works recursively: the rest of the program can be divided into a statement and the rest of the program etc.. However an expression (or statement) in isolation is only an abstract representation of an algorithm or part of an algorithm (it only possesses meta-meaning): to have any concrete meaning it must be interpreted with respect to some environment. This notion is similar to that of situation and situation-type meaning [Barwise & Perry 83]. This is the substance of a continuation: it is two objects: an expression and an environment [Strachey & Wadsworth 74]. In fact there are two types of continuation, command and expression. The former transfers control (or takes some sort of control decision) relative to a particular environment, and the latter evaluates an expression in the given environment (which may include some sort of control initiative). In the formal notation used in [Scott & Strachey 71]:

K: Env x Exp → Val **expression**

C: Env x Com → Val **command**

It shortly becomes apparent that all of the common (and indeed the not so common) control mechanisms can be cast semantically in terms of continuations. In the same way that looping may be described using

recursion, recursion may be described with continuations. Coroutine and generator operations are closely related to the concept which is popularly called lazy evaluation [Henderson & Morris 76] [Friedman & Wise 76]. Suspensions and continuations are one and the same: that is an expression (command) and an environment in which to evaluate it. The process concept is a little harder to characterise. It depends on the position whence it is considered. From inside it appears to behave autonomously except where it is necessary to have some interaction with another process or protected resource. Externally it is much as a coroutine, but is forcibly suspended and resumed by a scheduling process (no this is not recursive!) controlled by interrupts. Therefore it is amenable to the same analysis as a coroutine. This can be seen as a problem in resource management where the user processes are queued up (selection and prioritisation are a separate matter) to use the CPU (the protected resource). More detailed work is required to ensure fairness and prevent starvation (see Chapter 8). An alternative view is the idea of engines [Fitch 82] [Haynes & Friedman 84]. A process is given a time slot and told to run until complete or the time is exhausted (when it returns a continuation). This relies on the process yielding control as appropriate and so may create difficulties in attempting to prove the properties above with regard to fairness and prevention of starvation.

Having shown how continuations seem to form a complete foundation for programming, it should also be said that they are not necessarily always the most efficient way to provide a particular feature e.g. the looping constructs. That is a consequence of the features offered by most hardware rather than indictment of continuations themselves. As is often the case in this subject the practice is in advance of the theory.

Reflections on binding

The previous section attempted to make a case for why continuations are useful; if that premise is accepted, how can they be provided more efficiently than is apparently the case at present? The research presented in this thesis is concerned with the representation of the access environment. The question of the management of the control environment is only discussed in passing for a variety of reasons:

(i) it is only of secondary importance. Expression continuations can handle all but a few of the problems to which multiple environments are applicable (although a command continuation may be more efficient). Expression continuations are usable in a single control environment system, but command continuations require the existence of corresponding multiple access environments.

(ii) the problem has already been solved to a greater extent by the spaghetti stack model [Bobrow & Wegbreit 73] (that is not to say it does not warrant further investigation).

The rest of this section considers how present binding schemes work and shows how an analysis of them lead to the foundation of the new model.

The advantage of deep binding is fast context-switching while the strength of shallow binding is fast variable interrogation: can a reasonable compromise between the two be found? Since variable lookup is more frequent than context switching, any potential solution must be weighted so that this cost is negligible when compared with the shallow binding cost. One way of viewing deep and shallow binding in relation to

one another is that they are at opposite ends of a spectrum of binding models. At one end deep binding keeps all its associations in a single list. At the other end shallow binding keeps each association separately. This is in fact simplified to a spectrum of hashing functions: the deep scheme has a very simple function, the answer is always the same, that is it always hashes to the same bucket. The shallow scheme is in effect a perfect hash: each value hashes to a unique bucket, that is the identifier to which it has been bound. As one moves from one extreme to the other variable lookup becomes easier and context switching harder, because the environment component is progressively filtered out of the lookup process.

The last remark leads to another interesting relationship between the accepted binding methodologies and also puts the new scheme into context. The variable lookup process can be expressed abstractly using denotational semantics as before. Thus deep binding can be considered as

$$V: Env \rightarrow Id \rightarrow Val \qquad (1)$$

Then $V(e)$, $e \in Env$, is a function which takes an identifier and returns a value. In implementation terms the function V is uncurried to give a function (called ASSOC) and takes an alist (the environment) and an identifier as arguments.

By a similar analysis shallow binding is expressed so:

$$V: Id \rightarrow Val \qquad (2)$$

The shallow binding scheme has no environment information, hence the interrogation function and its semantics are very simple. The exact means of discovering the value is implementation dependent, but it is generally done by indirecting through the identifier.

It was remarked earlier that binding lookup is more frequent than context switching, so that although the latter must be efficient enough to permit more than passing use of it, the primary concern is speed of lookup. The key to a particular value is the identifier that is used to name it. The lookup process can be likened to form a lazy β-substitution in that the value of the free variable is 'substituted' into the computation when it is needed. Shallow binding keys one value to a name in the current environment. Therein is its strength and its weakness. The value is strongly connected to the name, but has no context - the environment is modified, extended and discarded, but only the changes are recorded. Deep binding on the other hand relates a value and a name within a particular environment, so the environment takes precedence over the value. Put another way the value is connected more strongly to the environment than to an identifier. Environmental extensions are truly that: a new set of bindings (i.e. the extension) is constructed and joined to the existing environment - the whole structure is designed for manipulating environments. Variable lookup is only a secondary consideration.

The main conclusion to be drawn from the above is that to find a new binding mechanism whose costs are weighted in favour of variable lookup, it will be fruitful to consider a method where the values are associated directly with the name, whilst environmental information is of lesser importance. To express this concept in semantic terminology:

V: Id → Env → Val (3)

This is the converse of deep binding. V(i), i ∈ Id, is a function which takes an environment as an argument and returns a value. The above function is the key difference between this and other binding models. The question now is how to identify an environment, and how to pick the correct value for the current environment from the set of values which may be bound to an identifier. This is the problem which was hinted at in the previous chapter: now, unlike lexical scoping, the context of a reference cannot be determined in advance. A particular binding of a free variable can be interrogated in any environment that is a descendant of the environment in which it was bound. The one piece of information that is known is the binding enviroment of the identifier: to find the correct value from amongst the set of possible values one needs to know which values were bound in environments which are ancestors of the current environment. The solution to this is an heredity function. The specification is that given two environments, it returns true if the first environment is an ancestor of the second, and false otherwise.

The question of how to resolve this last matter is dealt with in detail in Chapter 2. Having established the theoretical basis for the model, Chapter 3 builds on this and describes the development of the implementation. This is followed, in Chapter 4, by a discussion of the implications for the compiler. The hope is that the new model should show performance similar to shallow binding for simple programs and yet have the flexibility of deep binding for continuation programs. In order to observe the behaviour several other binding models have also been implemented: deep, deep with the function cell trick, non-cached version of the new model and continuous rerooting. These are analysed

statically (complexity analysis) and dynamically (benchmark programs) in Chapter 5. Chapter 6 addresses a side issue: the provision of local variables in the interpreter as a natural consequence of the new mechansim and more generally how the environment labelling technique can be used as a method of scoping arbitrary objects. Chapter 7 considers the effect of multiple environments on the garbage collection process. The important point here is how the correct data structure can have critical consequences for the algorithm and overall efficiency. Finally, in Chapter 8, some consideration is given to how continuations are useful, focussing on the areas of symbolic algebra, databases and the implementation of objects.

Chapter summary

This chapter describes the key insight which lead to the majority of the work presented in the rest of this thesis, namely, the idea of trying to find a way to associate a particular binding with an environment and a set of these pairings with an identifier. Deep binding exemplifies the inverse of this concept by associating the identifier and the binding and then grouping sets of these to construct environments. It seems reasonable to suggest that the only other method of representing information about multiple environments is the technique adopted in full shallow bound systems (e.g. INTERLISP-10 and systems which support the rerooting operation), that is, to record all the changes that have taken place in the environment. The approach adopted here has not, to the knowledge of the author, been tried before. In addition it would seem that these three approaches comprise the total number of ways of attacking the problem since if a variable is to have different values in different environments, there are only three possible forms for the interrogation function as laid out on the second section of this chapter.

CHAPTER 2

Ancestry Functions

Tree labelling

The evaluation process generates a tree of environments, where the nodes of the tree are environmental extensions. Can a suitable method be found to label such a tree, so that by comparing the respective labels of two enviroments a relationship can be deduced? Desirable properties of a label (or tag) are

(i) finite representation, that is the space for storing the tag to be known in advance

(ii) extensibility, that is the labelling can be continued consistently from what was previously a leaf

(iii) cheap comparison

The labelling problem obviously lies in the province of graph theory, but the author has been unable to find any suitable algorithms either in the seminal work [Aho *et al.* 76] or in recent papers. Most research in this field has been directed toward solutions of the nearest common ancestor (nca) problem [Maier 79], [Sleator & Tarjan 83] and [Harel & Tarjan 84]. This is stated as: given two nodes x and y find their nca. The problem given above is in some sense a simplification of this: given two nodes x and y, is nca(x,y)=x? The dynamic nature of the tree could create difficulties for the published methods. The cost of the best of the above schemes [Harel & Tarjan 84] is still unacceptably

high for the intended application at

O(n+mlogn) per query, where n=node count and m=operation count

Of greater concern is the amount of space taken. It is classified as O(n), but each vertex, v, maintains a list of all the children of v in the same ply (their terminology) and an array of all the ancestors of v in the same ply. This means that updating the relations of a vertex is O(n). Consequently it has been necessary to develop new algorithms specifically to solve this problem [Padget 83].

There are several schemes for labelling trees, although not all of these satisfy all of the criteria set out above. All of the methods considered will be explained in the order in which they were discovered. For simplicity the descriptions will be limited to binary trees. The straightforwardness of the extension to n-ary trees varies considerably.

The bit string

The first scheme is largely due to Ian Holyer. The root node of the evaluation tree is labelled with zero. For each left branch taken a zero is appended to the string, and for each right branch a one is appended. It is obvious that it is simple to continue the labelling from a leaf in a consistent manner, for the algorithm described above still applies. To show the ancestry relationship between two tags it suffices to check that the shorter label is a substring (starting at the beginning) of the longer label. An example of the mechanism in use is given in Figure 2.0. Unfortunately it falls short on two counts: the space for storing a label cannot be determined *a priori*, and indeed could become quite large. Secondly, the cost of comparison has no upper bound for the same

reasons. If the length of the bit string cannot be predetermined neither can that of the substring; hence the cost of testing may vary widely. Code is as follows:

```
(de ancestorp (env1 env2)
  (cond
    ((greaterp (stringlength env1) (stringlength env2))
     nil)
    (t (stringcompare
         env1
         (stringslice 1 (stringlength env1) env2)))))
```

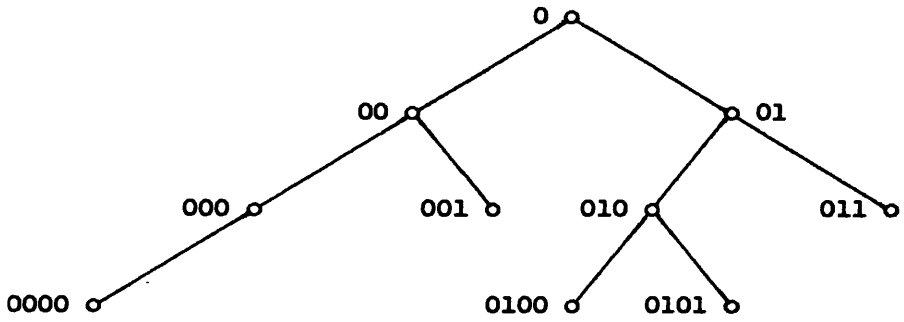
In order traversal

The second scheme follows from a suggestion by Robin Sibson. Each label consists of a pair of numbers constructed during an in order traversal of the tree. Let there be a counter, whose initial value is zero. As each arc is traversed (in either direction), the counter is incremented. On the descent, the value of the counter at each node is taken for the first element of the pair. On the ascent, the counter value is taken for the second element of the pair, thus a labelling such as in Figure 2.1 may be obtained. This looks more promising; the size of a tag is always two integers. The comparison of two labels is straightforward: given ℓ_1 and ℓ_2 we wish to discover whether ℓ_1 is an ancestor of ℓ_2; that fact is supplied by the expression

first element of $\ell_1 \leq$ first element of $\ell_2 \leq$ second element of ℓ_1

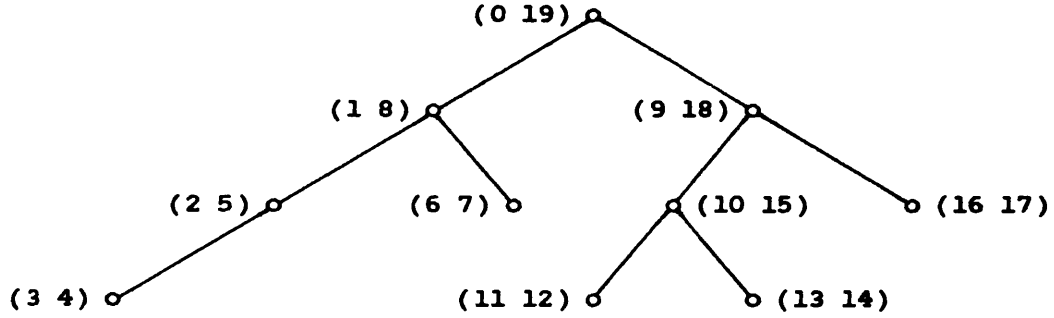
The major drawback is that the method is static. It is not possible to extend a leaf and yet preserve the information needed. Code is as follows:

```
(de ancestorp (env1 env2)
  (and
    (greaterp (sequence env2) (sequence env1))
    (leq (sequence env2) (span env1))))
```



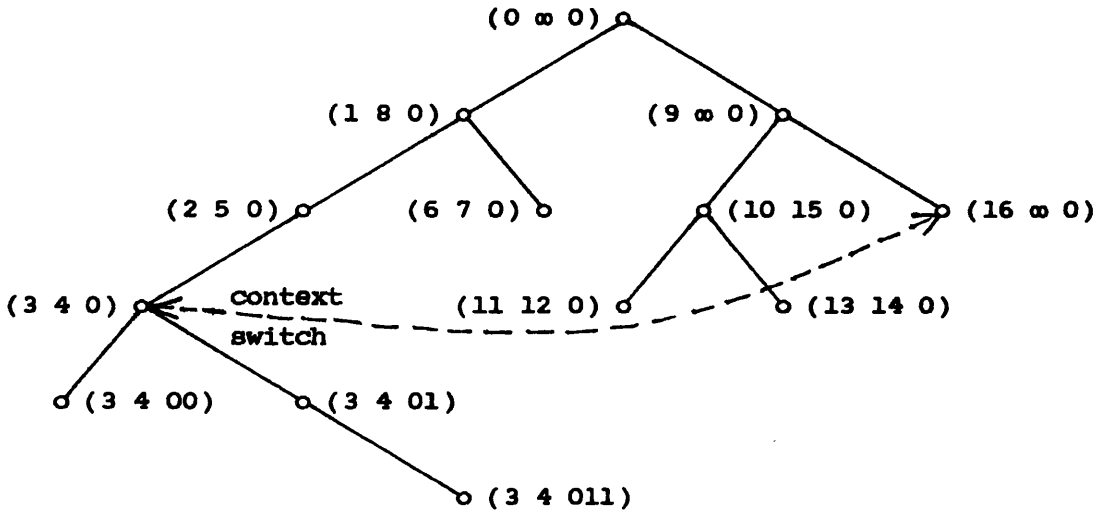
The bit string labelling scheme

Figure 2.0



The in-order traversal labelling scheme

Figure 2.1



The composite labelling scheme
(also showing the effect of a context switch)

Figure 2.2

A composite method

As the section title suggests this is based on a combination of the bit string and the in order traversal schemes. The evaluation tree is labelled as described above, but to overcome the extensibility problem the bit string is brought into play. Working on the assumption that the extent of continuations of previously preserved environments are relatively small, that is the tree is not likely to grow very much, it should be efficient in terms of both space and speed to use the bit string to establish ancestry. The basic label of each environment in the extension is that of the root of the particular subtree, which is where the new growth is rooted. The third element of the tag is given by the bit string of the intermediate root. When new labels are created it is this third part which is modified as described in the first section. An example of how the labelling works in this case can be seen in Figure 2.2 where there has been a context switch from (16 ∞ 0) to (3 4 0) and back. The heredity test now takes on two stages:

(i) compare first element of ℓ_1 with first element of ℓ_2 and second element of ℓ_1 with second element of ℓ_2. if these are both equal then ancestry is proven by comparing the bit strings as described in the first subsection.

(ii) perform the subrange test of the previous section

An example of this labelling scheme is given in Figure 2.2. Code to implement this is given over:

```
(de ancestorp (env1 env2)
  (cond
    ((eq (extension env1) (extension env2))
     (and
       (greaterp (sequence env2) (sequence env1))
       (ieq (sequence env2) (span env1))))
    ((greaterp
       (stringlength (extension env1))
       (stringlength (extension env2)))
     nil)
    (t (stringcompare
         (extension env1)
         (stringslice 1
           (stringlength (extension env1) (extension env2)))))))
```

The weakest part of this idea is the assumption made about the depth of a given tree extension. There are two specific criticisms:

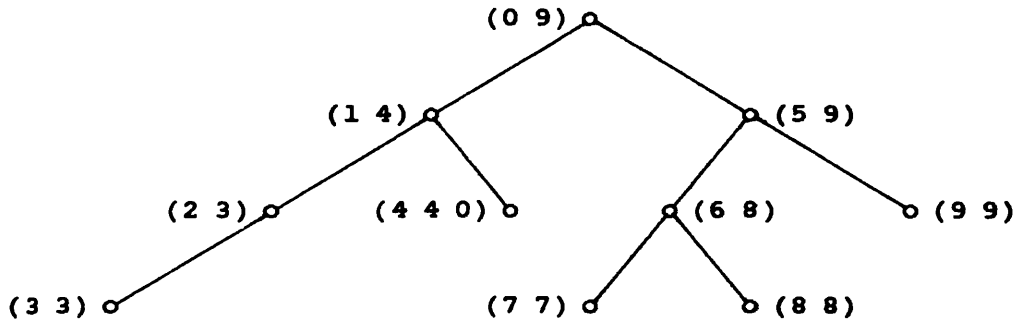
(i) if the bit string tag is implemented by a single machine word (or even two or three etc.), the mechanism is severely limited. Inevitably a case will arise with which the scheme cannot cope.

(ii) the alternative is to implement a general bit string data type and attendant functions. This must all but lose the efficiency which is the sole reason for pursuing this scheme.

In conclusion, this particular form of the ancestry predicate seems to create difficulties whichever way it is implemented.

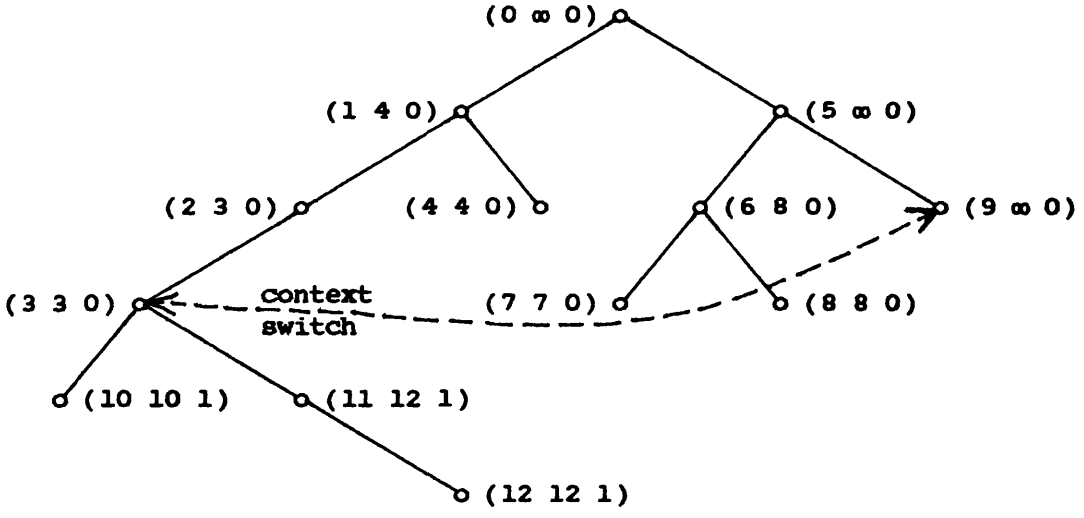
The chosen method

This scheme is both a refinement and an extension of the second one. Rather than refer to the first and second elements of the label the names sequence and span will be used. Sequence denotes that a node is the nth node created since the root node. Span denotes the highest sequence number that occurred in the tree below that particular node.



The refinement of in-order labelling

Figure 2.3



In-order with generations
(also showing effect of context switch)

Figure 2.4

First the refinement: this is implicit in the note about span above. Observe that it is the sequence number that is used to check the subrange and therefore that it is unnecessary to increment the counter when ascending during the traversal. For a graphic representation of this see Figure 2.3. The heredity test remains exactly as before. The worth of this change is that the count only increases by

$$\Omega(\text{total number of nodes})$$

rather than

$$\Omega(2 \times \text{total number of nodes})$$

Further optimisations are feasible in practice, but since such detail would obscure the discussion, their description will be omitted.

Of course this still does not solve the extensibility objection. The solution changes the tag from a pair to a triplet, the third element being called the generation. The generation is an enumeration of each of the subtrees that grows from the initial tree. This enumeration can be controlled by another counter. When it is desired to extend the tree from a previously closed leaf, the counter is incremented and its value taken to form the generation element of the tag. In the more frequent case of expanding an open leaf, the generation is inherited from the previous node. As for the original basis of the tag, that is the span and sequence, these are constructed as before. Again an example of the labelling process after a context switch is given (see Figure 2.4). There has been a context switch from (9 ∞ 0) to (3 3 0) and back. A new definition of the heredity predicate is needed; the tests are as follows:

(i) if generation ℓ_1 is greater then generation ℓ_2 as well as revealing that the nodes are in different subtrees it also show that ℓ_1 cannot be an ancestor of ℓ_2. Why? Because a new subtree can only ever be built on top of a tree of lower generation, which is in turn because the generation counter can only increase.

(ii) if the generations of the tags are the same (which implies they are in the same subtree), then heredity is given by

$$\text{sequence } \ell_1 \leq \text{sequence } \ell_2 \leq \text{span } \ell_1$$

(iii) thus generation ℓ_1 must be less then generation ℓ_2. The subtree containing ℓ_2 is connected to a tree of an earlier generation by its root and only by its root. Therefore if ℓ_1 is an ancestor of ℓ_2 then ℓ_1 must be an ancestor of root ℓ_2, which is in a previous generation, hence the algorithm may start over at case (ii) above

An example of this labelling is given in Figure 2.4 and here follows the code:

```
(de ancestorp (env1 env2)
  (cond
    ((eq (generation env1) (generation env2))
     (and
       (greaterp (sequence env2) (sequence env1))
       (leq (sequence env2) (span env1))))
    ((greaterp (generation env1) (generation env2))
     nil)
    (t (ancestorp env1 (root env2)))))
```

This now has all the properties outlined at the beginning of this chapter:

(i) representation is finite: the space required to store the label is three machine-sized integers

(ii) extensibility: by adding the idea of generations, computation can proceed from what were previously leaves in the evaluation tree, whilst keeping the tagging consistent

(iii) comparison of tags is relatively cheap: when both tags are in the same subtree, only a subrange test is needed. Otherwise, assuming the first tag is in a subtree closer to the root than the current subtree, then there is the additional cost of finding the appropriate ancestor of the second tag to compare against

In the next chapter details of LISP systems in which this method has been incorporated are given.

Chapter summary

This chapter has addressed the question of how a tree may be labelled in such a way that a simple test can be performed to check whether given two nodes one is an ancestor of the other. Having established desirable criteria for both the test and the label some possible solutions are developed and contrasted, concluding with a description of the method judged to be most suitable for this application.

CHAPTER 3

Practical Considerations

Development of the Implementation

In order to collect as much empirical evidence as possible regarding the new binding model and on binding models in general, it was intended to produce three implementations built on different existing LISP systems, namely Cambridge LISP, Portable Standard LISP (PSL) and Yorktown LISP (YKTLISP). Only the first of these can be called a complete implementation. The major part of the work on PSL was done during summer 1983 at the University of Utah. Unfortunately it has not been possible to obtain access to any machine capable of running PSL in the meantime to permit the addition of optimisations as the research progressed or to complete the garbage collector which was the one matter left outstanding. Thus there are no results for that system, although the work is largely complete. Administrative problems have conspired to prevent even a start being made on a version for YKTLISP.

The majority of the work presented in this thesis is of a practical nature; in large measure the topic of the thesis is also practical. It is concerned with trying to find a better way to represent multiple environments in dymanically scoped languages. This has lead to a deeper comprehension of the relationship between the theory and practicalities of the problem.

The first solution to the multiple environment problem, deep binding, was not a consequence of a consideration of fundamental issues. It was done that way because it was the simplest way that the implementors of

LISP could see to achieve a form of, for want of a better phrase, on-the-fly β-reduction in the interpreter, whilst still providing a mechanism for higher order functions. In a search for improved efficiency (and for no higher motive) shallow binding was developed. In this way LISP was stripped of one of its most potent semantic features, the FUNARG.

By appealing to semantics a concise behavioural description of the two methods mentioned above can be given (see Chapter 1). Obviously to have any chance of supporting multiple environments, the variable lookup process must take advantage of some environmental information. The new proposal can be viewed as the converse of deep binding. The semantic expression for the environmental interrogation process has become:

V: Id → Env → Val

This means that the identifier is viewed as a function, which when given an environment as argument, selects the appropriate value. The bindings of an identifier can be regarded as a set of pairings of environments and values. An important *caveat* though is the need for an efficiently implementable environment labelling function which permits ancestry validation. Several possible solutions to this problem were described in the previous chapter, ending with the one chosen for experimentation. Now that the tools for building a system as outlined have been formulated, work can begin.

Cambridge LISP

Cambridge LISP was chosen to be the vehicle for the major part of the research presented here, because of availability, a suitable machine on which to run it and local expertise [Fitch & Norman 77]. The system is programmed in two languages. BCPL [Richards 69] is used for the interpreter and base system. The rest, that is utilities and packages such as the compiler, reader, printer, editor, break etc. are coded in LISP itself. Over the period of the project the system has undergone tremendous changes, brought about either by a need for parameterisation or improvement in the BCPL support environment.

In the early stages the LISP sources and the ASLISP (Aquae Sulis LISP) sources were disjoint: it was quickly realised that this was a mistake and the systems were rapidly getting out of step with one another in the parts which were in fact common. The addition of a preprocessor permitting conditional compilation alleviated many of these problems. Now there is only one source, but it supports seven binding mechanisms and several combinations of operating system and machine. Although conditional inclusion was a big improvement, in many places the source was very untidy largely because of the differences in accessing and modifying variables between the various models. This has been solved by the most recent changes to the front end of the BCPL compiler which features manifest functions, which work by tree substitution. This means that the same expression may be used to access or update variables throughout the interpreter, but that it is macro expanded from a definition provided in an appropriate header file obtained by conditional inclusion.

The first target was an interpreter to support the new binding model. The following areas were identified as affecting this requirement:

(i) interrogation and modification of variables

(ii) the binding/unbinding functions (LAMBDABINDER and PROGBIND)

(iii) the allocation of space for and the initialisation of identifiers, known as *interning*

(iv) the garbage collector

The last matter is covered in detail in Chapter 7. Variable access is naturally a less straightforward process than the existing system which simply indirects through the pointer to the atom to find the current value. It also needs to maintain more information about each atom, and the logical place to keep this is with the atom. Accordingly the size of the structure has been increased. A graphic explanation is given in Figure 3.0. Because of the more complicated process involved in accessing and modifying variables two functions called INTERROGATE and MODIFY were written. These are now called via macros in the BCPL compiler. The names INTERROGATE and MODIFY are used everywhere in every system where LISP variables interact with the base code and macro-expanded to the appropriate form for the desired binding scheme. Some sections are completely different, such as the binding and unbinding of variables for lambda expressions and the program feature and the creation of atoms (MKATOM). Others are entirely new, such as the handling of closures in EVAL and APPLY.

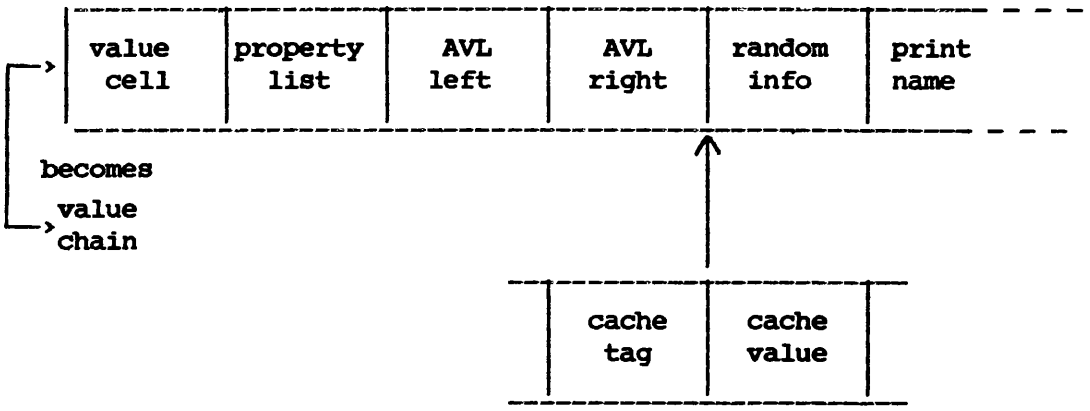


Figure 3.0

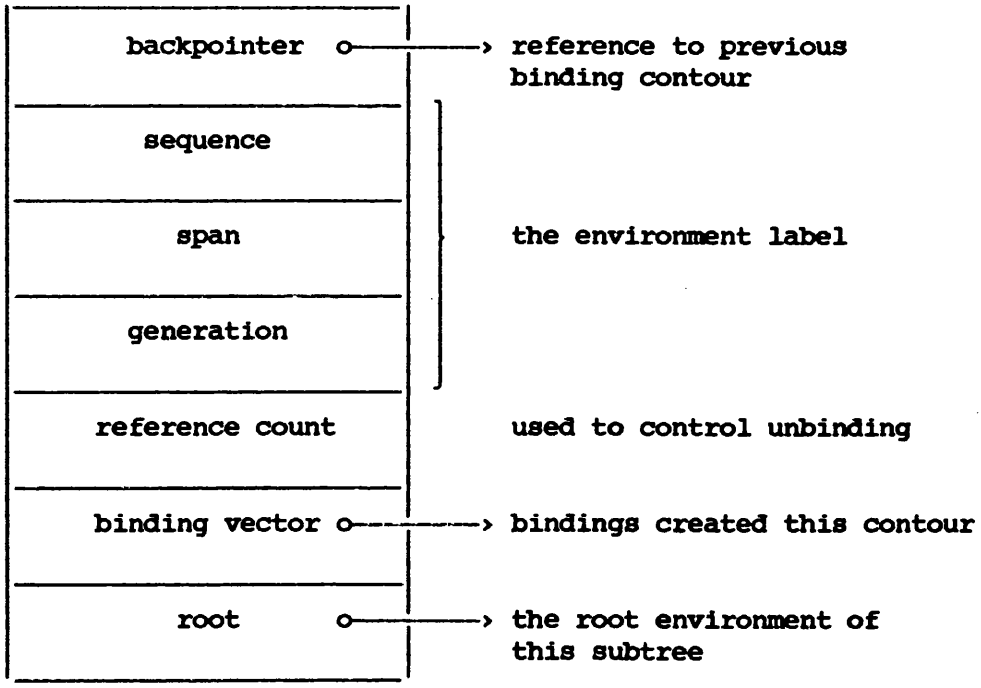


Figure 3.1

Binding and unbinding

The binding process has two sections:

(i) creation of a new *environment descriptor*

(ii) binding of the variables to new values

The environment descriptor is a vector of seven elements as shown in Figure 3.1. The value chain of an identifier can be thought of as taking the form of an association list in which the pairs are made of environment descriptors and values. To bind the variables the program descends the formal parameter list CONSing a new environment value pair on to the existing value chain of the identifier. For lambda expressions, the values are taken from the intermediate argument block. Program variables are, of course, set to NIL.

Unbinding engenders a little more work. It also depends on the reference count of the binding contour being exited. Given the existence of closures, it cannot be guaranteed that the binding (‹environment› . ‹value›) of an identifier will be at the top of the value chain, since other branches of the evaluation tree may have been preserved on top of it without requiring this binding necessarily remain. This situation could arise in the instance of a continuation being invoked from the main strand of the computation, preserving a new environmental extension of itself and then returning, where the same variable occurred in all three contexts. Hence a search is made for the appropriate pair and it is spliced out of the list.

Adoption of a more suitable data structure than the obvious one

described above pays dividends. A new datatype, called a *binding vector* was introduced to the system. This vector contains four entries for each variable to be bound (ie. the same amount of space as previously used by the CONS cell model). The first two are used to form the up/down links of a doubly linked list and the second two are used to hold the environment descriptor and the binding of the variable. The cache reference (see next section) points at this second pair of locations. The value chain can be searched by descending the CAR pointer (see Figure 3.2). The corresponding value is accessed by indexing off that location rather than taking the CDAR of the value chain entry. The advantage of this scheme is that binding and unbinding are both very simple. In the first case the code runs a finger down the binding list and the binding vector synchronously and modifies the up/down links of the existing binding chain. In the second case the the code runs a finger down the binding vector of the current environment undoing and patching up the up/down links (compare Figures 3.2 & 3.3)

Value and reference caching

To find the current value of a variable, the value chain is searched looking for the first environment value pairing in which the environment is an ancestor of the current environment. When found, the associated value is returned, otherwise the variable is deemed unbound. Similarly to assign to a variable, the value chain is searched using the samecriterion. When the appropriate pair is found, the value part is modified (eg. by RPLACD) to contain the new value. This is what happens in the the non-cached implementation of the new strategy (referred to as NC for brevity).

Early on in the implementation it was decided that the use of a

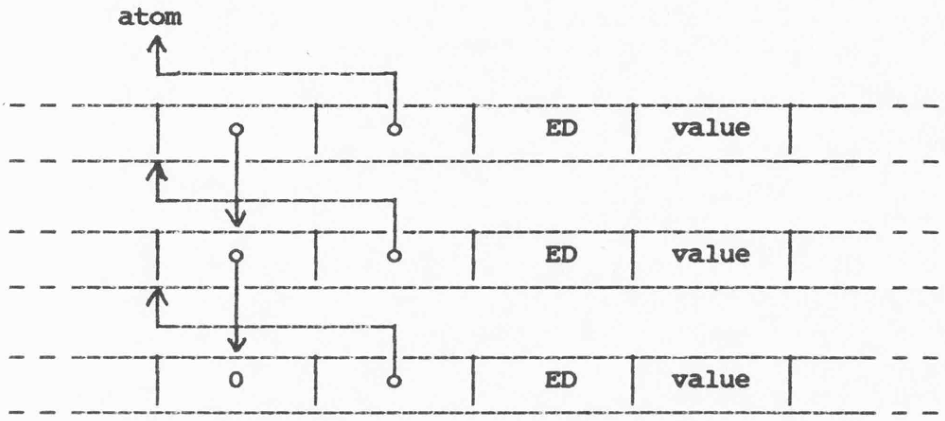


Figure 3.2

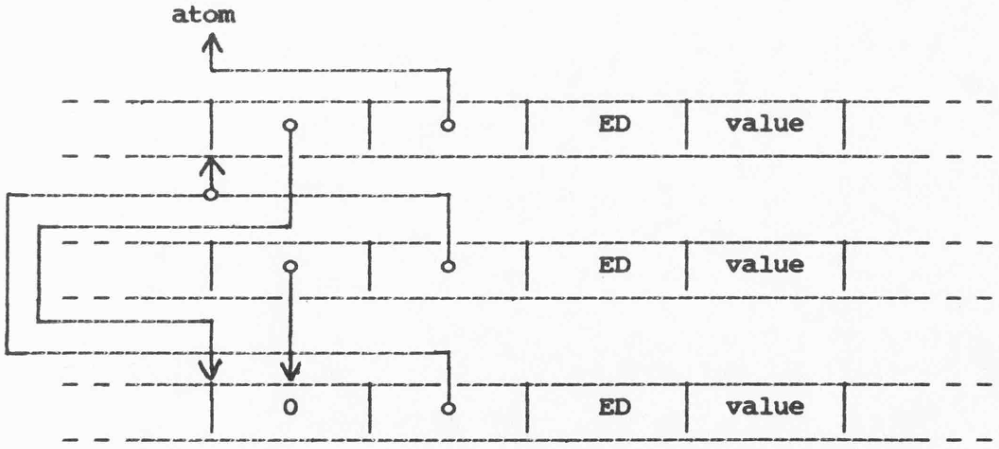


Figure 3.3

caching mechanism would be highly beneficial, and this was incorporated in the lookup/assignment strategy from the beginning. Thus on interrogation the value was also placed in the value cache, and the tag cache set to reflect the environment in which the variable had last been accessed. It is only possible to use an equality test on the cache tag to check validity. It is important to mention this since it might seem reasonable to apply the heredity test to the cache tag. That is not so, except in the case of no preserved environments! The reason for this surprising restriction arises from the case where a variable is bound in two environments, one nested inside the other, and the outer one has been passed down to the inner one. This might happen, say, in a mapping operation: the continuation would be applied to an element of the list being processed. That causes the cache of the variable to be loaded. If an heredity test were used to access the cache value, that binding would also be regarded as valid in the context in which the continuation had been *applied*.

The cache tag is also beneficial in speeding up the interrogation of variables which are bound in the global environment (such as functions), when there are many closures between the lookup and the binding environment. Recourse to the description of the ancestry test in the previous chapter shows that it must follow the chain of environments, through the root links, until one is found which is of the same generation. An important observation is that if one of the environments found during this search is equal to that held in the cache, then there cannot have been any new binding of the variable between the lookup environment and the one currently under investigation (the one that is equal to the cache). That means that the value in the cache is valid, and the search for further proof is unnecessary.

So far only the use of a cached value has been mentioned. That means variable access may take advantage of the cache, but that assignment must go through the more laborious process of full lookup to find the appropriate pair. That is potentially a great constraint on performance. It would be advantageous if assignment could also use the same cache. As has been observed before in computer science, an extra level of indirection solves a lot of problems. In keeping with this tradition, the above aim is achieved by caching a *reference* to the current (‹environment› . ‹value›) pairing, so that variable access must take the CDR of the cache for its result and modification may use RPLACD on the cache. This change, as well as making the system more consistent (by making the cost of access and modification equal), did make a significant improvement to performance (approximately 5% according to several benchmarks).

Instrumentation has shown that the caching is an important performance factor. In running various programs, such as compiling parts of the system, as well as some recognised test programs, the frequency of cache hit when interrogating fluid variables was never below 65%. Obviously the higher the frequency of a cache hit, the faster the program will run. Accordingly two possible sources of environmental perturbation, which would affect this hit rate, have been identified. These are garbage collection, which is discussed at length in Chapter 7, and descending across a fluid contour (i.e. unbinding).

In the naive implementation of the garbage collector the cached tag and reference were discarded and replaced by NIL. More recently, the tag is checked to see if it is an ancestor of the current environment and if so it is left untouched. Because of the state of the system it was not

possible to take direct measurements of how this improved the cache hit ratio, and more 'arms length' methods were employed, that is the running of benchmark programs. These showed 2-3% relative performance increases against the yardstick system (Shallow bound Cambridge LISP).

In the first implementation of unbinding only the value chain was modified to remove the (<environment> . . <value>) pairing: the cache was left untouched. This means, of course, that on the next interrogation of the variable (assuming it has not been rebound in the intervening period, which would also reset the cache), the cache would be invalid, and so a search is needed. Now the majority of programs are stack-like in behaviour, and hence, one can expect the binding of variables to behave in a similar way. Thus it is asserted that the next environment/value pair on the chain is *likely* to be the one sought. This will not always be the case, but it should happen sufficiently often to make the effort worthwhile.

So, on unbinding, the cache reference is loaded with the next pair from the value chain, and the cache tag is set to the binding environment of that value. It would not be correct to make the cache tag the environment being returned to, because there is no guarantee that the environment and that particular value are indeed related. Since the object of cache reloading is to impose as little overhead on the rest of the process as possible, a full scale lookup is not feasible. A simpler alternative is to set the tag to the one environment in which the value is known to be valid, that is its binding environment. It is not expected that this will be of more than marginal utility in the interpreter, with compiled code it is a different story. In interpreted code every LAMBDA and every

PROG creates a new fluid contour, whilst for compiled code there is a much greater gap between these boundaries (where the gap size may be measured in function calls), such that there is a greater liklihood of the reloaded tag and reference being valid. Tests indicated this was worth 3-4%.

Portable Standard LISP

Portable Standard LISP (PSL) is a descendant of Standard LISP [Marti *et al.* 79] and as such has a common heritage with Cambridge LISP. It is of course designed to be ported easily - it has not entirely achieved its aim in that respect (still some 3-4 man months are required to move the system to a new machine). Much work and research is currently being done toward that end. A major consequence of designing code for portability is that it is on the whole easy to modify, being divided up into functional sections, driven by parameterised expressions, and heavily sprinkled with macros. The changes were, as to be expected, in the same sections as listed at the beginning of this section. One particular difference between the systems is that PSL is entirely written in LISP, including the kernel, whereas it is in BCPL in Cambridge LISP. Therefore before even the first test could be run it was necessary to convert the compiler to the needs of the new scheme, rather than consider that as a later problem as was done in Cambridge LISP.

Being the second implementation, the process was much easier and faster, and it was possible in this system to take advantage of lessons learnt in the construction of the first one. A detailed description of the work involved would be tedious and unhelpful in the course of this thesis. For further information see [Padget 84].

There is one major difference between the philosophies (if they can be called such) of PSL and Cambridge LISP. It lies in the treatment variables and their evaluation. That is to say: in PSL the function position of a form is evaluated by a different technique from that used for the argument positions. This has no theoretical basis and is indefensible. The mechanism arose through a desire for greater speed in early LISP systems (see Chapter 0). Although the reason for its introduction has disappeared the anomaly remains. In the original deep bound LISP systems local variable lookup was reasonably quick. But there was a considerable price to pay when accessing 'global' variables, that is those bound at the outermost level, since in order to conform to the semantics an environmental search must be made first. Functions are very frequently bound at the outermost level and so the greater the depth of function call (and hence binding chain), the slower the system becomes. That can to some extent be alleviated by declaring an identifier global so the code to access them will go directly to the symbol table entry. The way taken to circumvent this lethargic (rather than lazy!) interpreter was to evaluate the function position of a form by *always* accessing the global value, whilst the rest of the form was treated in the canonical fashion. This is a dangerous semantic black hole because a name has different values depending on its position in the expression being evaluated. When shallow binding was developed there was no longer a discrepancy between the cost of lookup for local and global variables. Variable access became constant and therefore no reason remained to prolong the inconsistency. As remarked back in Chapter 0, expediency (and years of software effort) served to maintain the *status quo*.

A consequence of the above is that function lookup in PSL does not

use the scoping mechanism at all and preliminary results suggest it is marginally faster than Cambridge LISP. It is hoped that this implementation can be brought up to the level of the Cambridge system in the near future.

Yorktown LISP

Although no work has yet been done on this system the philosophy and approach taken to its design have not been without influence on the development of the Cambridge system [Blair 79]. In particular the strong theoretical foundation tempered with pragmatism served as encouragement to pursue solutions to the efficient provision of FUNARGS and highlighted the importance of being true to the mathematics upon which it is based. It is expected that a start will be made on YKTLISP during the summer of 1984.

Chapter summary

This chapter recounts the development of the two implementations of the new binding model and how dealing with the practical question of writing code to give physical form to an idea resulted in several improvements in the process. Although Cambridge LISP (CL) and PSL share a common heritage in Standard LISP they are very different systems; CL was designed and built with one machine in mind whilst PSL is intended from the beginning to be portable and easily modified. The impact of using the system as a base for experimentation in LISP implementation has been greater on Cambridge LISP, turning a non-portable program into a very much more portable one. The exercise also showed that PSL is not without shortcomings in ease of modification but recent developments in the latest versions should do much to circumvent these problems.

CHAPTER 4

Compilation and Closures

Compiler strategy

So far the changes have only been discussed with respect to the interpreter and the base system (although these are sometimes closely tied to the compiler as in PSL). The changes to the compiler are in many ways less implementation-specific because all LISP compilers have certain features in common, that is

(i) code to load and store free variables

(ii) code to bind and unbind free variables

Although there are other matters to consider such as register allocation, built-in assumptions and stack/argument allocation, the four cases above can generally be isolated and modified without interfering with the rest of the compiler. The compiler in the Cambridge system and the PSL compiler share a common heritage in the Portable LISP compiler [Griss & Hearn 81]; here the similarity ends. The first is a "one-and-a-half" pass compiler, while PSL takes a full three passes over the code. Once the relevant sections of each compiler were identified, producing new versions was relatively straightforward and so the matter is not dwelt upon in great detail. The method of the portable compiler is to convert the input expression into a list of macro operations for an abstract LISP machine (ALM). These instructions are then expanded by the machine dependent part of the program into code for a particular machine.

The Cambridge compiler

The macro operations in questions are !*store, !*load, !*fluidbind and !*fluidrestore. For loading and storing of fluids, the code generated checks the cache tag of the identifier and if it is valid loads the cached reference and then either takes the cdr of that, or rplacd's the new value, as appropriate, e.g.

* note that dm and dn are generic data registers

```
    move.l    0(nilr,dn.l),d7      indirect through quote cell
    move.l    16(z,dn.l),d6        load cache tag
    cmp.l     g-thisenv(g),d6      compare to current environment (ce)
    beq.s     *+10
    jsr       loadcache            make cache the value for ce
    move.l    20(z,dn.l),d6        load the cache reference
   [move.l    4(z,d6.l),dn         !*load ] load value
   [move.l    dn,4(z,d6.l)         !*store] store value
```

corresponding code for the shallow bound system is

```
    move.l    0(nilr,dm.l),dm      indirect through quote cell
   [move.l    0(z,dm.l),dm         !*load ] load value
   [move.l    dn,0(z,dm.l)         !*store] store value
```

The above code is for the Motorola M68000, but the spirit translates to other machines.

Binding of free variables is a little trickier because new store is required rather than saving the old values in the stack as was done in the original system; the need for store may cause the garbage collector to be invoked. The argument registers are not traced (and indeed would be destroyed by the GC) so these must be dumped into a safe region of memory. There they will be traced and relocated by the GC process, and can safely be reloaded afterwards. The compiler also assumes that binding does not perturb the argument registers, and so the binding routine must either avoid using them or save them whilst it does its job.

The support for binding fluids is general purpose in the sense that it

is utilised by functions with one, two, three or n arguments. The arguments are passed on the stack *and* in the registers. It is not possible to determine which of the argument registers contain legal values (e.g. in the case of a function with less than three arguments), so as a prelude to calling the binding code those which might not be valid are set to zero. In addition, because it might be necessary to invoke GC (which is a real function call, as opposed to the binding/unbinding process which is invisible) the framesize of the function requiring the binder must be loaded into a register so that the stack is moved up by the correct amount. The amount of store required for each fluidbind will vary proportionally with the number of variables plus a fixed overhead for the environment descriptor. This quantity can be computed easily at compilation time and so code is emitted to load another register with the number of addressing units of store that will be needed. Finally a third register contains the quote cell offset (also known as the SPID) of the fluid bind vector (historically known as the SPID-list i.e. a collection of variables to bind and the stack offset at which to store the SPID itself so that ERROR and CATCH can find it) e.g.

```
* the fluid bind vector is of the form
*     [ var1, var2, ..... varn, stack-offset]
* this is an example from a one argument function

    moveq.l   #0,d2              ensure legality of register
    moveq.l   #0,d3              ditto
    moveq.l   #framesize,d0      set framesize in case of GC
    moveq.l   #quote,d4          quote cell offset of vector
    moveq.l   #space,d5          amount of new space needed
    jsr       fluidbind
```

Unbinding of fluid variables only requires the quote cell offset.

The other significant change is that it is no longer necessary to allocate frame positions for each fluid variable in order to save the

previous binding. On the surface this merely seems a way to reduce the amount of stack used. It has a more profound effect though: if these locations were still allocated, but left untouched because there is no reason for the new model to use them, there is the potential for an illegal value to persist and cause mayhem in the garbage collector.

Various other minor improvements and modifications have been made to the compiler. In particular a function call where the value of the function was either held in a local, or was computed from an expression, the form being compiled was transformed from

$(exp_1\ exp_2\ \ldots\ exp_n)$

where exp_1 might be a local variable or another expression, into

$((lambda\ (!*unnamed!*)\ (!*unnamed\ exp_2\ \ldots\ exp_n))\ exp_1)$

where !*unnamed!* is a system defined fluid variable.

In effect this piece of code was substituted for the original expression. This is objectionable on two counts: an unnecessary new fluid contour is created, and, more importantly, there is the danger of name clash (albeit somewhat remote). An unseen consequence of this is that implicit application and the use of functions passed as arguments would impose an unsuspected overhead on the program. This has been overcome by providing extra support code entrypoints for these cases and making a few changes to the main part and the assembler stages of the compiler. Consider the following (pathological) function:

```
(de foo (bar baz quux) ((bar baz) quux))
```

This used to compile to (in ALM)

```
(!*alloc 5)
(!*stnil 5)
(!*stnil 4)
(!*store 3 3)
(!*fluidbind ((!*unnamed!*) . 4)
             ((1 !*unnamed!*) (2 baz) (3 quux)))
(!*store 1 (fluid !*unnamed!*))
(!*load 1 (!*reg 2))
(!*link !*unnamed!* 1)
(!*fluidrstr ((!*unnamed!*) . 4)))
(!*fluidbind ((!*unnamed!*) . 5)) ((1 !*unnamed!*)))
(!*store 1 (fluid !*unnamed!*))
(!*load 1 3)
(!*link !*unnamed!* 1)
(!*fluidrstr ((!*unnamed!*) .5)))
(!*dealloc 5)
```

and assembles to a total of 184 bytes (M68000)

and now compiles to

```
(!*alloc 4)
(!*store 3 3)
(!*load 1 (reg 2))
(!*link 1 1)
(!*store 1 4)
(!*load 1 3)
(!*linke 4 1 4)
```

and assembles to a total of 36 bytes (M68000)

There is a considerable saving in complexity, run time and code size.

The PSL compiler

The philosphy of this compiler is now quite different from its ancestor, because of the need to support SYSLISP (essentially an untyped LISP for writing systems level code) and the need to run on many machines. Both of these factors have led to a much more frequent use of macros in the first pass where the REFORM functions are applied (a detailed description of the phases of the compiler is not appropriate here, see [Griss & Hearn 81] and [Griss *et al.* 82]), so that operations may be tailored to specific machines. The intention of PSL is

to write the whole system in LISP (mixed with SYSLISP). There is no fixed code or assembler to interface with as in the Cambridge system. Experience has shown this is both an advantage because the concept is cleaner and a disadvantage because it is harder to isolate the mechanisms to change.

As before, the matters to be addressed are loading and storing fluid variables, and their binding and restoration. As part of the data-machine description from which PSL works, there is macro used explicitly in SYSLISP code to access LISP variables called LISPVAR (and an inverse PUTLISPVAR). When encountered in compilation

```
(LISPVAR <var>)      =>    (SYMVAL (IDINF <var>))
```

which is how the value cell of an identifier is accessed. SYMVAL and IDINF are compiler macros which expand to simple machine operations. IDINF strips the tag bits off its argument, and SYMVAL uses that value as an offset to indirect into a vector. This macro was changed to use the interrogation strategy of the new mechanism so that

```
(LISPVAR <var>)
    =>  (COND
            ((EQ THISENV (SYMTAG (IDINF <var>)))
             (CDR (SYMCHE (IDINF <var>))))
            (T (INTERROGATE <var>)))
```

The SYMTAG compiler macro uses its argument as an offset into a parallel array (containing the cached tag) to that used by SYMVAL. The macro SYMCHE, which holds the current cache reference, works in a similar way. If that fails the full lookup function (INTERROGATE) is invoked (cf. the example code generated by the Cambridge compiler shown in the previous section).

The system now knows how to compile common variable references - but not assignments. During the first pass in the standard compiler all references to non-local variables are embedded with the tag **$fluid**, so that a later pass will compile (SYMVAL (IDINF <var>)). This behaviour is very deep in the structure of the compiler, but is not what is wanted. Fortunately only one function creates the ($fluid <var>) or ($local <var>) or ($global <var>) objects (I&PANONLOCAL), hence this can be modified to return (LISPVAR <var>), which will expand as shown above. That is not entirely sufficient since I&PASETQ (the reform function for SETQ) uses I&PANONLOCAL. Normally the result of I&PASETQ would be of the form

$$\text{(SETQ}\ \begin{Bmatrix}\text{(\$local <var>)}\\ \text{(\$fluid <var>)}\\ \text{(\$global <var>)}\end{Bmatrix}\ \begin{Bmatrix}\text{(\$local <var>)}\\ \text{(\$fluid <var>)}\\ \text{(\$global <var>)}\end{Bmatrix}\text{)}$$

and the appropriate code could be generated. The above change to I&PANONLOCAL could result in I&PASETQ producing forms such as

```
(SETQ (LISPVAR <var>) <exp>)
```

which is not at all what is intended: trying to assign to the value of a fluid variable might have disastrous (as well as unintentional) consequences. Thus I&PASETQ must also be modified to recognise that the evaluation of its first argument must result in an L-value (not an R-value as immediately above). In particular if the variable to be modified is fluid I&PASETQ compiles this form:

```
(PUTLISPVAR <var> <exp>)
```

The second argument <exp> is expanded by I&PALIS which eventually

invokes I&PANONLOCAL (or even I&PASETQ) to resolve the expression. PUTLISPVAR will then be macro-expanded in a later pass so

```
(PUTLISPVAR <var> <exp>)

    =>  (COND
          ((EQ THISENV (SYMTAG (IDINF <var>)))
           (RPLACD (SYMCHE (IDINF <var>))
          (T (MODIFY <var> <exp>)))
```

The likelihood of having to garbage collect is catered for explicitly. The compiler is informed of the possibility by the use of a flag called I*UNSAFEBINDER. For a complete listing of the changes made please refer to the appendix following this chapter.

Compiler support for closures

The preceding two sections have described the specific changes made to the compilers to make them support the new binding model. This last part discusses the kind of additions which have been carried out to aid the efficient use of closures in compiled code. There are two ways in which a closure may be used:

(i) to perform a purely demand-driven style of evaluation (controlled lazy evaluation) which is provided by CONTINUE in the interpreter

(ii) to create high order functions (continuations), in which case the object is applied to arguments (done by APPLY in the interpreter)

it is overkill to compile these two uses as calls to the interpreter, i.e. EVAL and APPLY. They are very general functions, not really suited to such a specific purpose. Before describing a better strategy for handling

continuations in compiled code solution to this, it is useful to discuss how calls to the function CLOSURE itself are handled.

CLOSURE is a FEXPR function for the purposes of interpretation, so it does not evaluate its argument (quite reasonable given its usage). The argument to CLOSURE is an arbitrary LISP expression. It may be atomic, an expression or an anonymous function (ie. a LAMBDA expression). The reason for this generality is related to the two different applications of closure given above. If the expression designates a function, then closure is being used in the second sense, otherwise it is the first sense. The first sense can be regarded as a trivial function of no arguments and compiled thus, and so, in compiled code the two applications are unified. The current environment (the one that exists at the invocation of CLOSURE) is the closure of the function passed as an argument to CLOSURE (this name could lead to more confusion than FUNCTION!). The result of CLOSURE is a pair which is comprised of an environment descriptor and a function. This corresponds directly to the object described in Chapter 1 as an expression continuation. Compiled calls to CLOSURE are handled by a specialised function (closurel.compfn) which checks the form of the argument and then compiles it as a secret function. The link to CLOSURE itself is compiled open.

A new result is demanded of a stream (generator) by the function CONTINUE. In the previous paragraph it was explained how all continuations can be regarded as high order functions (albeit of no arguments on occasions). Thus

(CONTINUE <cont>)

can be compiled as

(APPLY <cont> NIL)

The assembler support for compiled code linkage tests to check that the function to link to is of the correct type: if it is not, then the situation is passed to APPLY in the interpreter. This is how compiled code calls interpreted code (or how errors are detected and signalled). Similarly an invocation of a continuation from compiled code would pass through this route. It is straightforward to piggy-back another test on to each of the entry points such that when the first comparison fails (ie. not a code pointer of the expected type), it next checks for a continuation, then if that fails passes on to APPLY. When the second check succeeds, a dummy frame is built above the current frame. This is used to hold the current value of the environment descriptor and a special return address where the context switch will be undone. The environment in the continuation is assigned to THISENV (the global variable which refers to the current environment). On return from the continuation control is passed to the address in the pseudo stack frame (which is a location in the fixed code) and the previous environment is reinstated. The purpose of building the frame is specific to this particular implementation. The frame is used to ensure that error recovery switches back the context as the stack is unwound.

The upshot of all this is that continuations are treated as first class objects by the system. They have the same status as functions. The implementation is still weighted toward the application of functions because that is always likely to be more frequent than the application of continuations. Continuations are a separate datatype with their own

distinguishing tag rather than being constructed from pairs with a special atom on the front and cannot be taken apart other than with the special selectors provided (CONTEXT - environment descriptor. BODY - function part).

Chapter summary

The nature and form of the changes to both the Cambridge and the PSL compiler were very similar. A particular problem to overcome was catering for the possibility of garbage collection during fluid binding. In the Cambridge compiler there is a strong built-in assumption that fluid binding is an almost invisible operation that takes place at function entry (and occasionally in the middle because of PROGs), which means it expects the state to be the same before and after binding. The potential need to garbage collect can make it diffcult to ensure that the state is preserved. The code to access, modify, bind and restore fluid variables is much more complicated in the new system (see example at the beginning of this chapter). An unusual usage of fluid binding arose in the existing Cambridge compiler when the function position of a form was neither a fluid variable nor a λ-expression. Reconsideration of this matter lead to much smaller simpler and more efficient code being produced. The compilation of continuations and their support has not created problems largely because they can be treated as special cases of functions.

A less obvious consequence of this chapter is that it has demonstrated that this quite complex model can be supported adequately on a standard architecture machine. In the longer term it is intended to implement this system on a particular machine called the Orion, which permits the addition of user microcode. Specifically, opcodes for fluid

load. fluid store. fluid bind and fluid restore will be written. This results in simplification of the program because all that had to be explicit for. say the Motorola 68000. will migrate down into the microcode and need no longer be the concern of the compiler.

Appendix to Chapter 4

The code shown below documents the changes made to the PSL compiler to support the new binding model. I&PASETQ, I&PANONLOCAL, I&PALAMBDA and I&PAPROG are all redefinitions of existing functions. FREEP, I&PAI_COULDBEFREEP and I&PAISI-JAP are additions.

```
PROCEDURE FREEP X; GLOBALP X OR FLUIDP X;

PROCEDURE I&PASETQ(U,VBLS);
% PA1FN: Convert (SETQ X1 Y1 X2 Y2 ...) to (SETQ X1 Y1)
%         (SETQ X2 Y2) in a PROGN.  Also check that Xi is a MEM
%         mode or $NAME
BEGIN SCALAR VAR, FN, EXP, LN;
LN := LENGTH CDR U
IF LN NEQ 2 THEN RETURN
<<  LN := DIVIDE(LN,2);
    IF CDR LN NEQ 0 THEN
    <<  I&COMPERROR LIST("Odd number of arguments to SETQ", U);
        U := APPEND(U, LIST NIL);
        LN := CAR LN + 1 >>
    ELSE LN := CAR LN;
    U := CDR U;
    FOR I := 1 STEP 1 UNTIL LN DO
    <<  EXP := IF FREEP CAR U THEN
                 LIST('PUTLISPVAR, CAR U,
                      IF FREEP CADR U THEN
                           LIST('LISPVAR, CADR U)
                      ELSE CADR U) . EXP
               ELSE LIST('SETQ, CAR U,
                      IF FREEP CADR U THEN
                           LIST('LISPVAR, CADR U)
                      ELSE CADR U) . EXP;
        U := CDDR U >>;
    I&PA1('PROGN . REVERSIP EXP, VBLS) >>;
% Should check that CONST's not SETQ'ed or BOUND
RETURN IF FREEP CADR U THEN
IF FREEP CADDR U THEN
    I&PA1(LIST('PUTLISPVAR,CADR U,LIST('LISPVAR,CADDR U)),VBLS)
ELSE
    I&PA1(LIST('PUTLISPVAR,CADR U,CADDR U),VBLS)
ELSE IF FREEP CADDR U THEN
    I&PA1(LIST('SETQ,CADR U,LIST('LISPVAR,CADDR U)),VBLS)
ELSE IF I&PAI-COULDBEFREEP(CADR U,VBLS) THEN
IF I&PAI-COULDBEFREEP(CADDR U,VBLS) THEN
    I&PA1(LIST('PUTLISPVAR,CADR U,LIST('LISPVAR,CADDR U)),VBLS)
ELSE
    I&PA1(LIST('PUTLISPVAR,CADR U,CADDR U),VBLS)
ELSE IF I&PAI-COULDBEFREEP(CADDR U,VBLS) THEN
    I&PA1(LIST('SETQ,CADR U,LIST('LISPVAR,CADDR U)),VBLS)
ELSE
<<  VAR := I&PA1(CADR U,VBLS);
    EXP := I&PA1V(CADDR U,VBLS,VAR);
    U := IF FLAGP(CAR VAR,'VAR)
         THEN LIST('I$NAME,VAR) ELSE VAR;
    IF (NOT (FN := GET(CAR EXP,'MEMMODFN)))
         OR NOT (LASTCAR EXP = VAR)
    THEN LIST('SETQ,U,EXP)
```

```
    ELSE FN . U . REVERSIP CDR REVERSIP CDR EXP >>;
END;

PROCEDURE !&PAI-COULDBEFREEP(X,VBLS)
% check to sdee if X might get declared automatically as a fluid
% deeper in the recursion - need to catch it now to preserve sanity
% in !&PASET:  I know the PROGN and T are not strictly necessary
% since !&MKNONLOCAL returns a non-NIL value, but the implmentation
% may change at some time, so play for safety.
ATOM X AND
NOT(ISAWCONST X OR
    CONSTANTP X OR
    MEMQ(X,'(NIL T)) OR
    NONLOCAL X OR
    MEMQ(X,VBLS)) AND
<< !&MKNONLOCAL X; T>>;

PROCEDURE !&PANONLOCAL(X,VBLS);
% Pass 1 processing of a non-local variable. The occurrence,
% embedded in an appropriate form e.g. ($LOCAL X), is emitted.
% The variable must be an established (declared) non-local
BEGIN SCALAR Z;
RETURN
    IF NOT IDP X OR NOT NONLOCAL X THEN
        PA1ERR LIST("non-local error",X)
    ELSE IF NOT !*SCANNING!-ARGLIST THEN <<
        IF FLUIDP X OR GLOBALP X THEN
            !&PA1(LIST('LISPVAR,X),VBLS)
        ELSE IF GET(X,'WVAR) THEN
            IF X MEMBER VBLS THEN <<
                !&COMPWARN(LIST('WVAR,X,"used as local"));
                LIST('!$LOCAL,X) >>
            ELSE LIST('WVAR,X)
        ELSE IF WARRAYP X THEN LIST('WCONST,X)
        ELSE PA1ERR LIST("Unknown in PANONLOCAL",X) >>
    ELSE IF FLUIDP X THEN LIST('!$FLUID,X)
    ELSE IF GLOBALP X THEN LIS('!$GLOBAL,X)
    ELSE IF GET(X,'WVAR) THEN
        IF X MEMBER VBLS THEN <<
            !&COMPWARN(LIST('WVAR,X,"used as local"));
            LIST('!$LOCAL,X) >>
        ELSE LIST('WVAR,X)
    ELSE IF WARRAYP X THEN LIST('WCONST,X)
    ELSE PA1ERR LIST("Unknown in PANONLOCAL",X);
END;

PROCEDURE !&PALAMBDA(U,VBLS)
% PA1FN: Pick up new LAMBDA vars for VBLS, check implicit PROGN
% Should maybe rename locals here?
<<  VBLS := APPEND(CADR U,VBLS);
    'LAMBDA . LIST(!&PALIS!-JAP(CADR U,VBLS,T),
            !&PA1(!&MKPROGN CDDR U,VBLS)) >>;

PROCEDURE !&PAPROG(U,VBLS);
% PA1FN: Pick up PROG vars, ignore labels.
<<  VBLS := APPEND(CADR U,VBLS);
    'PROG . (!&PALIS!-JAP(CADR U,VBLS,T)
        . !&PAPROGBOD(CDDR U,VBLS)) >>;
```

```
PROCEDURE !&PALIS!-JAP(U,VBLS,!*SCANNING!-ARGLIST);
% Sneaky support for new binder to let PANONLOCAL know who
% called it
!&PALIS(U,VBLS);

!*SCANNING!-ARGLIST := NIL;
```

CHAPTER 5

Performance

Performance in analysis and in practice

This chapter presents an informal analysis (a full complexity analysis is not possible because of the dynamic nature of the system), followed by timings for several benchmark programs on various systems. In particular it contains comparisons between systems with and without the new binding mechanism. This is an important test because it gives some indication of the change in performance created by the machinery to support full closures whilst running code which does not avail itself of the facility. In their existing guises, neither Cambridge LISP nor PSL provided any capability for environment capture, being based on shallow binding. It was therefore expected that the performance would be degraded: this was borne out in practice, but the difference was relatively small. It is hoped that the YKTLISP version, which uses deep binding with lookup caching for the access model and spaghetti stacks for the control model, will show an improvement over the existing system in line with the results presented in this Chapter. It is not sufficient simply to implement a new binding model and benchmark it against the existing shallow bound system. Certainly such tests show what price is being paid for a more general environment model when executing programs with stack behaviour, but the question of cost comparative to alternative multiple environment models is left open. To resolve this several competing strategies have also been implemented and tested, both on standard programs and those involving frequent context switch. These schemes are pure deep binding, deep binding with the function lookup modification, deep binding with cache cells, a non-cached version of the

new model, and environment rerooting.

Analysis

There is no point in presenting a cost breakdown for the new scheme without providing some information to judge it against. An informal analysis will also be given for shallow binding and deep binding which should demonstrate why it is reasonable to expect the new model to exhibit the efficiency of shallow with the potency of deep. There are three matters to consider in each case: variable access (and modification), context-switching and the binding/unbinding process. In the following sections these three questions are dealt with in turn. Binding and unbinding is split into a consideration of the two cases described in the next paragraph.

Comparing the cost of shallow against deep binding (and most other systems for that matter) is rather difficult, since the goals are hardly equivalent. Therefore in the first instance each system isconsidered with respect to a single binding stack to factor out the problems concerned with managing multi-headed (cactus) stacks and then a consideration of the more general multiple environment case follows. As always with this sort of performance breakdown there must be some level at which further overheads are disregarded: one such is the management of the binding stack (e.g. the cost of adjusting the stack pointer and the like). The manipulation of the fluid binding list is however included (see Chapter 4), because the differing formats are direct consequences of the particular binding mechanisms. For future reference the structures are as follows:

shallow - ((<variable> . <stack offset>)* . <stack offset>

the rest - [<variable>* <stack offset>]

The fluid binding list (SPID-list) is more complex for shallow binding since it also includes information about where to stack the current binding on entry and where to recover the previous binding on exit. The contents of the value cell of the <variable> are saved into the location at the paired <stack offset> away from the current frame base. Finally a reference to the list itself is stored in the last <stack offset> given in the list. The other systems have different techniques for saving the existing value of an identifier and none of these use the stack (indeed is this not the object of the exercise?). A reference to the SPID-list is stored at the specified offset in the alternative schema.

Deep binding

There are two parts to the environment as it is represented in this model: the global environment (or oblist) and the dynamic environment which may conceptually be regarded as an alist (association list). To discover the value of an identifier in a particular situation the alist is searched for the first occurrence of the identifier - the associated value is the one to use. If no pairing can be found, the value is taken from the entry in the oblist. Because the cost is determined by a dynamic structure, it is difficult to put hard bounds on the complexity. The best that can be done is to give an indication of the order of magnitude in terms of bindings. Of course this is also true for all the following discussions. So in the worst case, to find the value associated with a name cost is

O(# of bindings between here and the root environment + 1)

The 1 is to account for the global environment access. It is obvious from this why the function position evaluation strategy (or more accurately hack) is attractive, since this transforms the cost for interrogating some names to O(1). The cached version has the same worst case as pure deep binding but average cost should be lower. Since access and modification are symmetric, the costs above are equivalent for assignment.

Preservation of an environment is simply retaining a reference to the association after the current environment (CE) pointer has discarded it. Because *all* the necessary environmental information is kept on the alist, a context switch implies changing the CE pointer to refer to some other association list. Henceforth (until the next context switch anyway) expressions will be evaluated in the environment described by the alist. It seems it would be fair to describe the operation of environment capture as O(1).

Turning to (un)binding, the case of stack allocated environments is considered first. On entry to a new contour, the names and their new values are pushed on to this stack. On exit they are popped off. Therefore it seems that the cost is 3 memory operations (mops) per variable - 1 to access the element of the fluid binding list, 1 to push the name, and 1 to push the value. In total 3 x (# of variables).

In the second case the environment is heap allocated. A list is used to simulate a stack, so new pairings are added by CONSing, and bindings are lost by assigning the CDR to the environment pointer. This requires 2 CONS operations (to build and add the new binding) and 1 mop (to access variable) per variable for fluid binding. Each CONS

operation (cop) is assumed to require 2 mops. Unbinding is 1 mop per variable (ie. to take the CDR of the current environment). There is a 2 mop overhead for bind and unbind to reference the current value of the environment and to update the environment. In total (2 x (# of variables) + 2) mops + (2 x (# of variables)) cops which simplifies to (6 x (# of variables) + 2) mops. Note also the cost of the stack version of deep binding can still apply in a multiple environment model, for example INTERLISP-D.

Shallow binding

There is only one environment structure - essentially a global environment. By recording the changes in the environment, the global environment is used to hold values for particular environmental extensions. To access these values costs the same as to access identifiers known to be global in the deep binding model, that is O(1). The important point is that *all* identifiers can be accessed in O(1).

The story is somewhat less impressive with regard to context-switching. The environment structure is large and nebulous; there is no single object to reference which could facilitate the capture of the whole environment. Consider how the fluid binding/restoration process works in this model: for each identifier in the list of variables to bind, the value is taken from the value cell (i.e. the entry in the global environment structure), and placed on a stack. The value cell is then set to the new binding. When leaving a fluid contour, again using the list of fluid variables, the values are taken off the binding stack and replaced in the appropriate value cells, thus restoring the previous environment. This is a little more complex than deep binding.

The old values of the names are saved into the specified stack locations (refer to format and contents of fluid binding list shown in the first section of this chapter) and the new bindings are stored in the 'value cell' of the identifier. To bind requires 6 mops per variable (get variable/offset pair, get variable, get present value, get offset, store value into offset, set new value). Unbinding takes 5 mops (get variable/offset pair, get variable, get offset, recover value from offset, restore old value). In total these operations need 11 x (# of variables) mops. This is considerably higher than the purely stacking version of deep binding.

It is obvious that the only way to switch context is by repeated application of the binding/unbinding process. If the environment to restore is in a path from the current environment to the root then it can be done by unbinding. The general case of moving from one leaf of the evaluation tree to another is more complicated: first a node common to paths from each leaf to the root must be determined, then one can unbind down to this node and rebind up to the target leaf. The return journey is the reverse of this process. From this it can be seen that context switch is

O(# of bindings between source node and target node)

Only one major lisp implementation provides the FUNARG facility by this means: INTERLISP-10. The outstanding problem is how to discover the first common node. In INTERLISP-10 this is done by tracing down the dynamic chain from the target environment to the root, then starting the rebinding process from the source node until a marked environment is encountered, whence the upward trail commences having first unmarked

the chain.

An alternative solution (but one which is not fundamentally different), is described in [Baker 78b]. An outline of the method is given in the section on rerooting in Chapter 0. A more detailed version is now appropriate to show the costs involved. The scheme described above is a very complex form of context switch, but rerooting is a more accurate term and indicates how much effort is required. Because there is no single object which captures the state of the environment at one moment, context switching cannot be simply a matter of changing a single pointer. The environment is so deeply embedded into the evaluation mechanism that it cannot easily be abstracted; thus to move between relatively disjoint environments requires a drastic structural modification. The only information about the environment is the changes that have been placed in it, so the way to describe the relationship between two environments is to describe the differences between those two environments. Hence to move from one environment to another, those differences must be inverted (note that this operation is of course reversible). Consider for a moment the deep binding model: when at the root node of the evaluation tree, all variable references can be satisfied by accessing the global value cell. By binding one variable, the environment is now one step away from the global environment, and every variable lookup must check this single binding before the global value is taken. The effect of shallow binding is to move the root of the evaluation tree when binding or unbinding, so that the value of the global cell is always the correct one. Now notice that if the name is recorded with the value when changes are made to the environment a reversed deep binding list is constructed. This reveals two options:

(i) use this list to reinstate a previous environment (by exchanging the values in the binding list with the contents of the value cells)

(ii) regard the list as the association list, as in a deep bound scheme.

This ability to permit shallow or deep style variable interrogation is known as casual rerooting. There are some problems in the provision of the casual system over the continuous shallow bound version, since each free variable access must check what mode the system is in before committing itself to either indirecting through the global value cell, or searching the alist. This is much easier to support in a soft machine rather than one with a fixed instruction set.

The key to the rerooting system is two pointers, CE (current environment) and CE' (previous environment). To extend the environment a new CONS cell is created (which will be the new value of CE). An alist is constructed (by CONSing on to this cell, the entries of which are an identifier and the *current* value of that identifier (see Figure 5.0). When the list of formal parameters is exhausted, there remains one more identifier to save - CE'. In this way, for each new fluid contour created, CE' is rebound, so that on unbinding to the previous environment, CE' recovers *its* previous value, which is of course the preceding previous environment. The binding for CE' is created in a slightly different way: the cell which *was* referenced by CE is RPLACAd to point to the pairing (CE' . <value of CE'>), and then RPLACDd to point to the alist constructed in the first part of the process (see Figure 5.1). Now the binding is complete.

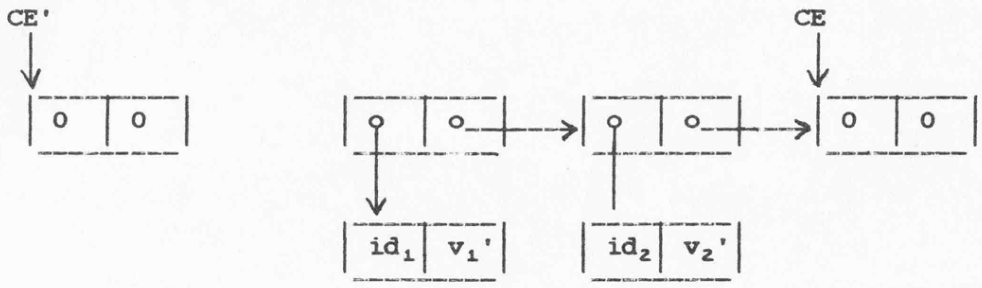


Figure 5.0

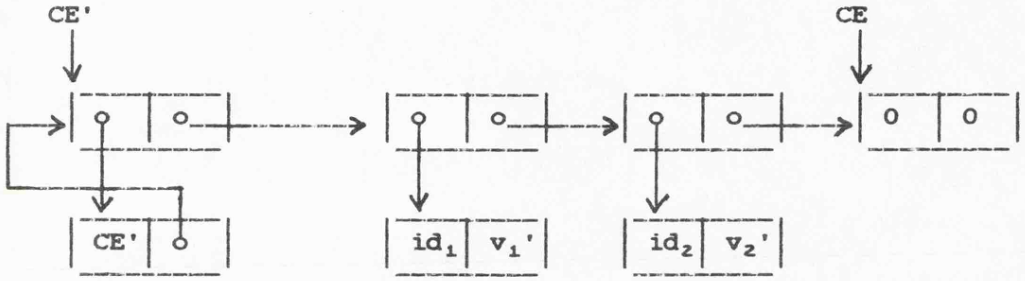


Figure 5.1

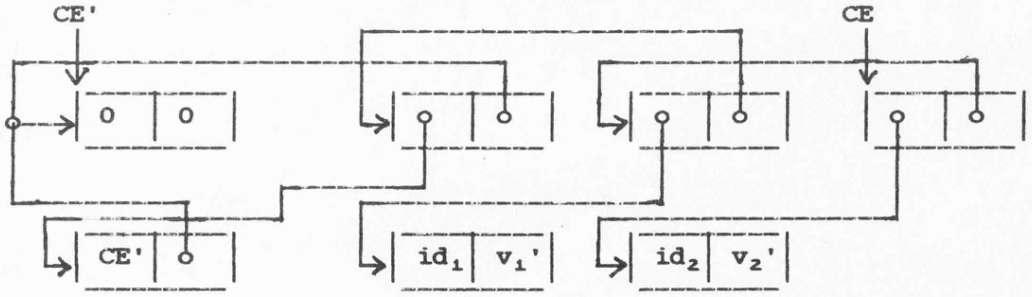


Figure 5.2

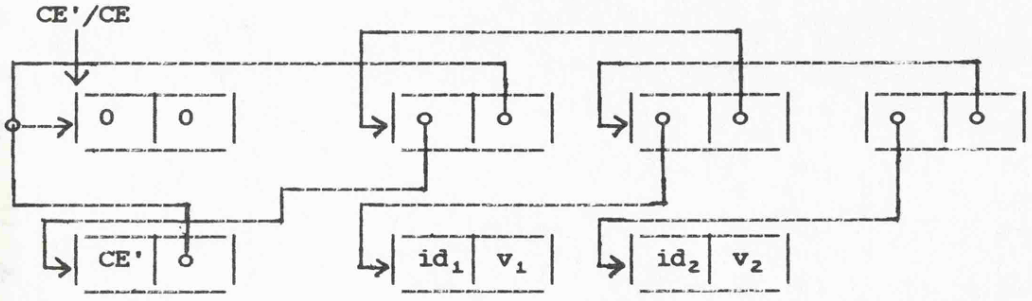


Figure 5.3

The restoration process takes two passes: the first reverses the alist between the source and target environments (see Figure 5.2), then in the second stage this list is traversed exchanging the contents of the bindings in the alist with the contents of the respective value cells of the identifiers (see Figure 5.3). The reversal stage is the crux of the model. The list *must* be reversed in the case of a general rerooting (such as when moving the tree more than one fluid contour from its current position), so that the environmental entropy is completely inverted for the second pass. The second reason for the importance of the reversal is that after traversing the list undoing the modifications, that list, when viewed from what was CE, now encodes the extension from the target environment in the style of a deep bound system. It is this feature which permits *casual* rerooting. The user is given access to a primitive which changes what happens at rerooting, so that variable accesses can be resolved either by the shallow lookup method or by ASSOC.

What is the cost of rerooting? By reference to the diagrams mentioned above, binding requires (2 x (# of variables) + 2) cops (one each for new CE and the alist entry for CE', and two for each variable bound) and 2 x (# of variables) + 6 mops. These six break down into one to access value of CE', one to reset CE', one to access present value of CE, one to reset CE, two to set the CAR and CDR of the old CE pair, one to access the present value of each variable and one to set the new value. This totals (6 x (# of variables) + 10) mops.

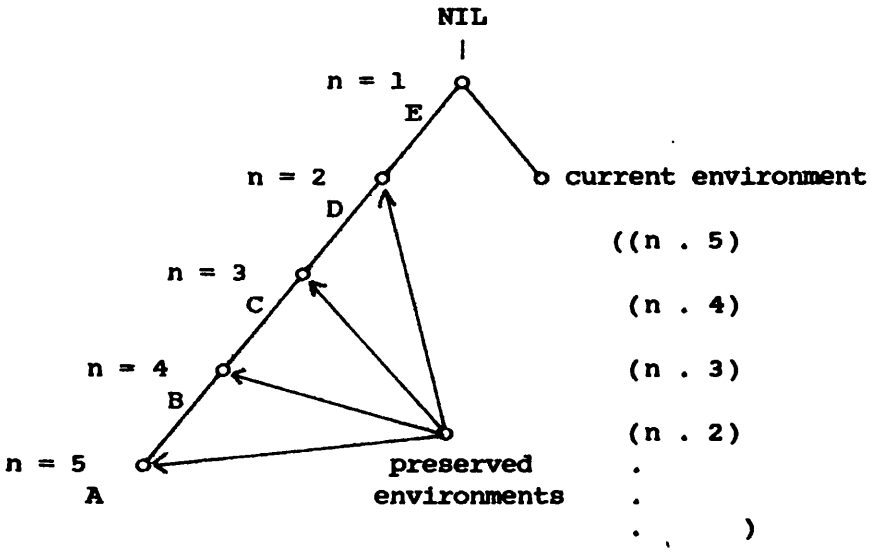
The new model

The binding or rather bindings are still distributed like shallow binding, but they are related indirectly by an environment descriptor which is created at each new fluid contour. The atom structure contains

a list of pairings of environments and values. The association of an environment and a value indicates that the identifier has that value in the specified environment, and by virtue of the inheritance property of the labelling scheme, the identifier will have that value in all environments descended from the one in which it was bound, unless it be rebound. By searching this list, the first environment value pairing which satisfies the test that the environment is an ancestor of the current environment is taken to be the current value. Thus the cost of the search is

O(# of bindings of a particular variable)

In general it is reasonable to assert that this will be less than the total number of bindings, which is the cost of deep binding. This appears to be borne out in practice (especially running single context programs); see the timings for the non-cached version later in the empirical results section. When the closure feature is not being used, the current binding will always be the first pairing on the chain. However, if many closures including different bindings of the same variable exist, the cost of lookup could be considerably higher for that particular variable than for deep binding. The point at which that cost outweighs the considerable overhead of full environmental search for *everything* as in pure deep binding is very difficult to determine and will depend largely on the dynamic behaviour of the program. Although the complexity bounds (worst case) stay the same for each system, this may not work out in practice. This anomaly is best explained with reference to a diagram (see Figure 5.4). It can be seen that in the deep bound system, for each of the environments A through D, the variable n is the first binding encountered. For the new model on the other hand, because the environment does not have a direct reference to the


NIL
n = 1
E
n = 2
D
n = 3
C
n = 4
B
n = 5
A
current environment
((n . 5)
(n . 4)
(n . 3)
(n . 2)
.
.
.
)
preserved
environments

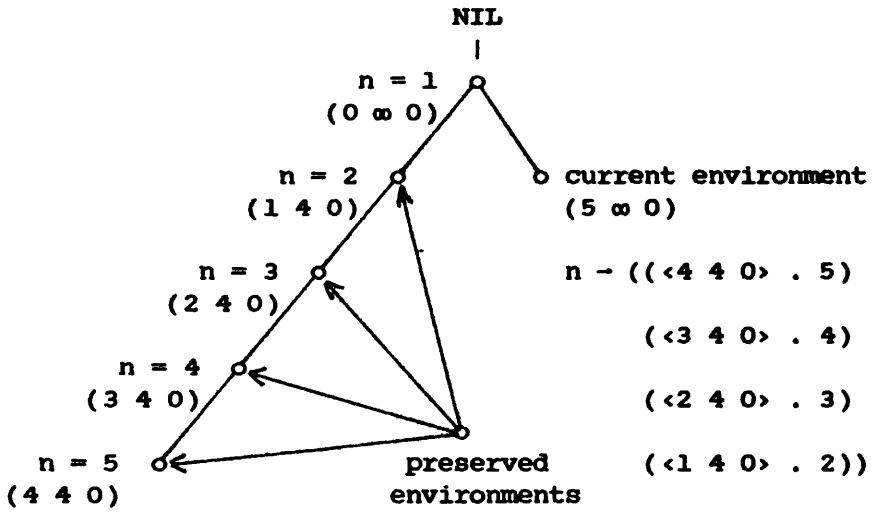

NIL
n = 1
(0 ∞ 0)
n = 2
(1 4 0)
n = 3
(2 4 0)
n = 4
(3 4 0)
n = 5
(4 4 0)
current environment
(5 ∞ 0)
n → ((<4 4 0> . 5)
(<3 4 0> . 4)
(<2 4 0> . 3)
(<1 4 0> . 2))
preserved
environments

Figure 5.4

particular binding in that environment, the search is potentially much longer. For instance, the value in environment <1 4 0> will not be found until after four comparisons. This can be reduced in the cache version of the model by using the fact that if the cached tag is equal to one of the environment descriptors found during the search, then no binding can have taken place between the current environment and that environment, and therefore the cache is valid.

The context switching operation is very simple, akin to deep binding. Because all the environment relationships with respect to the current environment can be established by the environment label only the current environment descriptor need be changed. Henceforth all variable interrogations will be carried relative to the new environment, so context switch itself is O(1).

There are two stages to the binding process:

(i) construction of a new environment

(ii) the binding of the variables in the new environment

The first part builds a vector, inserts the new environment label and sets the reference count, the back link and the root pointer (root of current subtree). That is 8 mops (an extra one is required to make the vector header). The second stage entails 8 mops (one each to set environment pointer and value, one to access name in binding list, one to get existing value chain, two to set the upward and downward links of the new chain entry, one to set the upward link on the previous binding, and one to set the new value chain).

There are two possible strategies for unbinding:

(i) leave the bindings with the variable and only reset the current environment pointer (similar to deep binding when using a frame structure rather than just an alist)

(ii) remove the bindings associated with this environment from the value chain

The first option is simple, but although it would make this operation very fast, the effect on performance later may be undesirable because the correct binding will move further and further down the environment chain as control moves closer to the top level.

The second is more complex but worthwhile. In the first method it does not matter whether the environment has been preserved or not, the bindings are still extant. If they are no longer reachable, eventually the garbage collector will remove them. In the second case it is necessary to know whether the bindings have been saved. The simplest method of maintaining this information is by adding a reference count to the environment descriptor. So the operation to remove the bindings is conditional on this value. In the present version of the system the count is actually only used as a flag to control unbinding. In a later system it may be feasible to use the count to collect downward FUNARGs on exit from that frame (see also the section on future work in garbage collection in Chapter 7). To unbind a variable, the particular environment value pair created on entering this contour has to be spliced out of the value chain. The environment descriptor contains a pointer to the binding vector, and so unbinding is a simple matter of running down

this structure resetting the up and down links of the entries (binding vectors) above and below the contour being left (see Figure 3.3). This means the cost of unbinding is 4 mops per variable (plus 1 mop overhead to access the binding vector from the descriptor).

The effect of caching

As mentioned in Chapter 3, in order to reduce the frequency of need for search, value caching was incorporated into the scheme. This is only an implementation optimisation and not fundamental to the model. It can equally well be applied to a deep bound system with similar results. Two additional slots are associated with each atom. These are used to keep the cached value or reference and the environment in which that reference is valid. When an identifier is first bound the environment and reference cache are changed to be valid in the current environment. In general however the first interrogation of an identifier is likely to be of the order given previously for each scheme. Subsquent enquiries in the same environment will be satisfied in bounded time, that is O(1). The operations are:

(i) access environment cache

(ii) compare with current environment

(iii) take cached value/reference

(iv) take/set value in reference

The above discussion mentioned both value and reference caching. In the first implementation which included caching, only values were

saved. As well as making SETQ a very much more expensive operation, it is also complicated in that the cache and the value chain entry must be updated. Such a strategy would preclude (or at least greatly complicate) the open-coding of assignment. Subsequently reference caching was adopted, wherein a pointer to a location containing the current value is saved. This has the advantages of trading one extra indirection for a unification of the access and update operations. Hence SETQ takes the same time as a variable access (see Chapter 4 regarding compilation), and can be open-coded in a similar manner. Tests indicated a performance improvement of approximately 10%-15% as a result of this change.

Empirical results

The foregoing sections have now set the scene for the actual results of testing and running the various systems. As always benchmarking systems is a difficult process, open to both question and abuse. Before presenting the tables of timings it is pertinent to describe what sort of tests were carried out, the scope of the individual tests, how general the results may be claimed to be and how the figures should be interpreted.

There has been a growing interest over the past two years regarding the relative merits and efficiencies of various LISP systems. Consequently a large number of programs have been developed and widely distributed in an attempt to reduce the comparisons to a numerical value. A potential criticism of such programs is that they are not truly representative of applications programs and the figures thus indicated will not be relevant in practice. For this reason some other, one hopes more general, figures were recorded of the time taken to compile several system modules, then a soak test was made in the guise of building the

REDUCE algebra system [Hearn 83]. This was followed by running the REDUCE test file which is a suite of programs designed to display the abilities of the system; as such it exercises many of its facilities, so it could be regarded as a representative problem set, which gives reasonable credence to the timing results obtained from it. Several of the recognised "benchmark" programs exhibited severe deficiencies, such that in some cases they were not measuring the features intended. Appropriate remarks accompany the description of the offending programs.

The test programs are taken from four sources: a suite now known as the Griss tests which were collected by Martin Griss as a means of comparing the relative efficiencies of various PSL implementations, a theorem-prover developed by Boyer and Moore and the REDUCE system and its test file. In particular, there is one feature that all these programs have in common; they do not use FUNARGSs, therefore the only conclusion that may be drawn from the running of these tests is speed relative to a shallow bound system.

The efficiency of the system supporting the new model has been charted frequently during its development to demonstrate the advantages or disadvantages of various modifications as they were made. These are presented and discussed first.

The systems are:

(i) deep binding (using alists)

(ii) deep binding with function cells

(iii) shallow binding

(iv) continuous shallow binding with rerooting (using alists)

(v) new model

(vi) new model with caching

What are the justifications for believing that the tests run and the figures returned are a true representation of performance in practice? Why were these tests chosen?

In moving from the Griss tests, through the Boyer test to compilation and REDUCE, the programs are becoming more and more general. In the first instance the tests exercise fairly specific areas of the system so serving two purposes: to measure performance of the dominant feature, and to highlight deficiencies. The other benefit of running recognised tests is that figures exist for many other systems, so comparisons can be made. In the end however, it is the speed of execution of 'user' programs that matters. The difficulty is to find a representative 'user' program. The only hope is to take a very large program with the intention that sheer size, generality and unpredictability will combine to give a fair measure. For this reason the REDUCE algebra system was selected. The first task is to compile this. Distributed with the REDUCE system is a test file which is part example of the capabilities of REDUCE and part system work-out. This then uses many features of REDUCE, so although in itself it might not be considered as a user program, its components are. Timings exist for the execution of this on several systems and so an impression of relative performance is revealed.

The optimisation of the implementation of the new model

This section is not so out of place in this chapter as the heading might suggest. The benchmark programs did much to highlight inefficiencies and areas for further work during the development of the system. In the first instance, as with any new idea, the highest priority was to solicit the desired behaviour from the system, and direct effort at performance improvement later. In keeping with this plan all the peculiarities related specifically to the new model were identified and coded in the base system which is written in BCPL. In this way it was fairly easy to ensure that the correct algorithm was being applied although it was only of moderate efficiency. These changes were sufficient to generate an interpreter which satisfied the aims of the new model. The next step was the modification of the compiler as discussed in detail in Chapter 4.

There are five areas in which the needs of the compiler impinge on the new model:

(i) loading of fluid (common) variables

(ii) setting of fluid (common) variables

(iii) binding of fluid (common) variables

(iv) restoration of fluid (common) bindings

(v) function calling

Again initially the emphasis was on functionality abetted by caution.

In the existing shallow bound model, these services are provided by open coding in the first two cases and by pieces of fixed code (written in assembler) for the rest. This technique, the register allocations for calling the fixed code and the perturbation of certain registers over the call are fundamental to the structure of the compiler, so it was inconceivable to change them. Thus in the first incarnation the assembler mechanism was simply used as a linkage mechanism to the existing BCPL code. As was to be expected the figures were somewhat slow (see Table 5.0).

	shallow	new/I	new/II	new/III	%
emptytest 10000	220	220	240	240	-
slowemptytest 10000	1660	4240	2180	2120	27.7
cdr1test 100	2280	2260	2660	2660	16.6
cdr2test 100	2320	2260	2660	2660	14.6
cddrtest 100	1520	1520	1800	1780	17.1
listonlycdrtest1	13600	13600	15880	15860	16.6
listonlycddrtest1	18120	18040	21280	21160	16.7
listonlycdrtest2	13540	13580	15860	15780	16.5
listonlycddrtest2	18120	18020	21280	21220	17.1
reversetest 10	1760	1740	2040	2000	13.6
myreverse1test 10	1760	1740	2040	2000	13.6
myreverse2test 10	1700	1680	1940	1920	12.9
lengthtest 100	3560	3780	4460	4460	25.2
arithmetictest 1000	8780	10040	10020	9960	13.4
evaltest 10000	17720	25100	22320	21420	20.8
tak 18 12 6	3940	3900	4480	4460	13.1
gtak 18 12 6	14820	29140	18640	18320	23.6
gtsta g0	67080	80060	81140	78780	17.4
gtsta g1	67460	80340	81520	79200	17.4
mkvect 1000	320	440	360	360	-
getv 10000	340	1880	480	480	41.1(*)
putv 10000	360	1900	500	500	38.8(*)
checked getv 10000	900	2440	1140	1120	24.4(*)
checked putv 10000	960	2500	1200	1160	20.8(*)
getv local 10000	280	-	320	320	-
putv local 10000	300	-	340	320	-
checked getv local 10000	840	-	980	960	14.2
checked putv local 10000	900	-	1020	1020	13.3

Table 5.0

As the system was developed and improved this set of tests were run again and again (whilst recognising their deficiencies) to show what changes, if any, had been wrought. To put these figures in context a

complementary set of figures for the shallow system and the final percentage differentials are also given. In the initial stages only the Griss suite was used as a performance measure, because it did not seem worth running anything more complex until most bugs and inefficiencies had been removed. Thus Table 5.0 charts the coarse honing of the system.

In some cases version I is faster than the existing system and than any of its successors. This is because early versions of the compiler did not produce checking of car/cdr access which is normally the default. Many of the tests show no marked difference between the systems; this is to be expected when there are no fluid variables in use and no binding. New/I is the preliminary version of the system, where assembler was only used as a linkage mechanism to BCPL code; new/II has full assembler support; new/III is the most recent version, which has the revised data structure (originally the value chains really were alists) and several optimisations on the cache operation. Blanks indicate that the difference is not worth measuring. The starred (*) times above are of little or no consequence. The length of the test is so small as to prevent a statistically significant result. Also note that clock resolution is fairly coarse giving a latitude of ±60msecs.

The Griss tests

These are made up of 24 small programs, each one evaluated many times inside a loop to give a measurable time. The majority of the tests show little or no difference between the systems, and indeed this is to be expected in many cases, since the problems do not exercise anything to do with the changes that have been made (except for function calls). In many cases what may be written as a function call will be compiled open

(e.g. IADD1) and so the test executes within a single body of code.

Table 5.1 shows the results of running the Griss tests on all of the different binding schemes.

	X (cache)	X	reroot	shallow	deep	deep (cheat)
emptytest 10000	240	260	220	220	240	240
slowemptytest 10000	2120	2960	1620	1660	3160	1840
cdr1test 100	2660	2660	2360	2280	2680	2640
cdr2test 100	2660	2640	2380	2320	2680	2640
cddrtest 100	1780	1780	1580	1520	1820	1780
listonlycdrtest1	15860	15840	13960	13600	16020	15800
listonlycddrtest1	21160	21080	18940	18120	21460	20900
listonlycdrtest2	15780	15860	14460	13540	16020	15780
listonlycddrtest2	21220	21100	19480	18120	21460	20900
reversetest 10	2000	1940	1780	1760	2020	1940
myreverseltest 10	2000	1940	1740	1760	2040	1940
myreverse2test 10	1920	1880	1680	1700	1940	1880
lengthtest 100	4460	4460	4060	3560	4500	4440
arithmetictest 1000	9960	10360	8920	8780	10660	9820
evaltest 10000	21420	26400	20420	17720	45660	26960
tak 18 12 6	4460	4440	4020	3940	4500	4400
gtak 18 12 6	18320	22860	14800	14820	24280	16500
gtsta g0	78780	90480	76340	67080	139660	77880
gtsta g1	79200	90820	76720	67460	140020	78280
mkvect 1000	360	400	300	320	420	340
getv 10000	480	940	340	340	1020	980
putv 10000	500	960	360	360	1020	1000
getv check 10000	1120	1560	900	900	1680	1600
putv check 10000	1160	1620	960	960	1740	1660
getv local 10000	320	320	300	280	320	300
putv local 10000	320	320	300	300	320	340
checked getv local	960	940	880	840	980	960
checked putv local	1020	1000	900	900	1040	1000

Table 5.1

There are some significant results amongst these:

(i) emptytest and slowemptytest: the first uses small integer arithmetic, so that it contains no other function call and everything has been open compiled; the second uses generic functions and therefore must use fluid lookup for the function, thus providing some indication of the cost of fluid interrogation

in comparison with the yardstick system.

(ii) lengthtest involves a function call to length, but note that the relative differences for the various systems are quite small because length itself is recursive and the compiler applies an optimisation in the self-call in this case.

(iii) arithmetictest provides almost exactly the same information as (ii), except that a recursive factorial is substituted for length

(iv) evaltest is interesting in that it does give a reasonably true indication of the relative speeds of the interpreters. The relevance of this figure depends on the split between interpreted and compiled code.

(v) tak and gtak provide one of the more significant results, because although it is recursive and hence the self-call optimisation is incorporated, it does involve frequent calls to generic functions in the case of gtak.

(vi) gtsta g0 and gtsta g1 is rather similar to the evaltest in that the results hang on the efficiency of the interpreter, and so should be seen in the same light.

(vii) array access tests. In some respects, these are the most misleading of all - obviously a different binding mechanism does not affect the time to access an element of a vector, but there is a fairly large disparity in the figures. This is because it becomes a test of the cost to access a fluid variable (the

vector) plus the cost of a generic function call (PUTV/GETV). For this reason additional tests were made using a local binding of the vector.

Some of the above tests seem to indicate that the non-cached version is occasionally faster than the cached version; one factor in this is clock resolution (which is between 20, 40 and 60 msec observed). It is also known that performance varies with memory size (a consequence of wait states, clocking and backplane speed).

The figures for the non-cached version are interesting since they show the cost of the basic interrogation scheme. As might be expected rerooting is about the same as shallow binding (note that none of the tests actually involve binding fluid variables). Deep binding is provided because that has equivalent functionality to the new model; although the implementation is simple, the price is quite high. Deep binding with a different lookup method for the function position serves two purposes:

(i) to show how the cost of function lookup dominates the cost of deep binding

(ii) a reasonable comparison with the method employed in INTERLISP

The Boyer test

The code for this test is generally available, and is therefore not given. It is part of a theorem prover developed by Boyer and Moore [Boyer & Moore 79] to analyse a restricted class of LISP programs for validity. It does not seem to offer a very general indication of

performance, because the program structure is somewhat unusual. It is recognised as having a very high frequency of function call and CONSing operations. The most heavily used function accesses and sets a fluid variable (profiling reveals it is called approximately 500,000 times). This variable is strictly global since the test is semantically flat and there is no fluid binding or unbinding.

Shallow	87.22
X (cache)	94.60
X	124.38
Rerooting	87.26
Deep	128.16
Deep (cheat)	95.71

Table 5.2

Compilation

Speed of compilation is an interesting case to consider, not only because the compiler is a reasonably large program making fairly heavy use of fluids and fluid contours, but also because it is dependent on the form of the program being compiled. This last feature should, given a variety of inputs, mean that the results are generally applicable. Times are given for building four packages which form part of the programming environment of Cambridge LISP.

(i) reader and pretty printer

(ii) avl tree builder and manipulator

(iii) structure editor

(iv) disassembler

Finally, there are figures for building the REDUCE algebra system

which is made up of 68 modules and about 30000 lines of code.

	X (cache)	X	reroot	shallow	deep	deep (cheat)
In core	1431.48	1544.64	1160.50	1144.20	-	-
Compiler	714.64	741.64	574.60	620.74	-	1984.74
Readprint	108.22	122.98	83.82	100.22	223.30	151.06
AVL-tree	25.86	29.23	19.48	23.60	52.38	35.14
Disasm	57.56	72.36	47.42	52.90	130.78	90.00
Editor	149.34	184.10	120.42	139.92	332.20	230.58
REDUCE	3297.28	-	2941.90	2882.16	-	-

Table 5.3

Reduce test

The justifications for this last test were outlined in the prologue to this section and need not be reiterated.

Shallow	177.70
X (cache)	182.26
X	-
Rerooting	170.50
Deep	-
Deep (cheat)	-

Table 5.4

Speed of context switch

This leaves but one area uninstrumented: the very feature which is the object of this work, namely context switching. It is rather difficult to find any general programs to test this facility because so few widely available systems provide it. In fact the only implementations to support it properly are YKTLISP and INTERLISP. Some small tests have been devised, the results from which should probably be viewed in the same light as those for the TAK function. The programs are Eratosthenes' sieve for generating prime numbers, an example of possibilities lists, taken from the INTERLISP manual (1978 edition p12.18), a program to produce continued fraction approximations to square roots and a tree-walking function to solve the same-fringe problem. The latter

program is discussed in some detail in Chapter 8 in the section on applications. The sieve was used to generate the first 100 prime numbers, the possibilities list to produce the first 2000 Fibonacci numbers, the square root generator to produce the first 20 approximations for all the numbers with non-integral square roots up to 100 and the tree-walk was applied to compare two trees of 8192 leaves with equal fringes. The code for some of the above is given in an appendix to this chapter.

	X (cache)	X	reroot	deep	deep (cheat)
Sieve	23.12	28.78	173.68	169.54	12.72
Fringe	20.34	22.62	34.92	25.52	15.36
Possibilities	7.64	9.32	8.54	11.98	9.62
Cont. Frac. sqrt	7.90	10.94	8.90	14.82	10.00

Table 5.5

Chapter summary

The two parts of this chapter assess the analytical and the practical measurement of several binding mechanisms. The results seem to indicate that the new model varies between 5% and 15% slower than the original shallow bound Cambridge LISP system. This is better than all the other models bar rerooting which is to be expected since in the face of stack-like behaviour rerooting is equivalent to shallow binding. There are several other benchmarks which have been collected into a suite by Richard Gabriel. It is intended to try to obtain these programs to conduct further tests in the near future. Comparison of cost of context switching is not so comprehensive but those examples which have been run show the new model to good advantage and have been surprisingly bad for rerooting.

Appendix to Chapter 5

(i) Sieve of Eratosthenes program.

```
(de from (n) (conz n (from (add1 n))))

(de filter (p x)
     (cond
          ((zerop (remainder (car x) p)) (filter p (sdr x)))
          (t (conz (car x) (filter p (sdr x))))))

(de sieve (l) (conz (car l) (sieve (filter (car l) (sdr l)))))
```

which is invoked by (sieve (from 2))

(ii) Tree walking program. See section on tree searching in Chapter 8.

(iii) Possibilities list program.

```
(dm possibilities (gfn)
     (subst gfn 'gfn '((lambda (posslist) (poss1 gfn)) nil)))

(de poss1 (gfn)
     (closure
          (prog (tmp)
               (cond
                    ((null posslist)
                     (cond
                          ((eq (catch nil (gfn)) 'finished)
                           (return 'finished)))))
               (setq tmp (car posslist))
               (setq posslist (cdr posslist))
               (return tmp))))

(dm au-revoir () '(throw 'au-revoir nil))

(dm adieu () '(throw 'adieu 'finished))

(de note (x)
     (cond
          ((null posslist) (setq posslist (ncons x)))
          (t (nconc posslist (ncons x)))))

(de fib (f1 f2)
     (closure
          (progn
               (note f1)
               (note f2)
               (setq f1 (plus f1 f2))
               (setq f2 (plus f1 f2))
               (au-revoir))))
```

which is invoked by (setq foo (possibilities (fib 0 1)))

(iv) Continued fraction square root approximation. See section on algebra in Chapter 8.

(N.B. CONZ is a form of CONS which suspens its second argument)

CHAPTER 6

Generalisations of environment labelling

This chapter stands on its own: alone it can be viewed as a justification (at least in part) for the work described in the other chapters, or as an adjunct to the main theme. The primary task is to show the serendipitous benefits of the environment labelling approach and then to discuss how this can be used to clarify the semantics of LISP systems.

Lambda calculus and scope

Lambda calculus did not contain any decree as to the scoping of variables: it was not necessary. Indeed the concept is meaningless in that context because the calculus is a reduction language. There is only one sense in which the language could be said to be (lexically) scoped; that is after all α-reductions and all β-substitutions have been made to remove free variable references. If LISP is to be lexically scoped, then this view demands that programs either be represented as vast single expressions or be split into manageable fragments with no free variables. Put another way, this avoids β-substitutions - but at the price of making them the reponsibility of the programmer. He must simulate the effect by passing such variables down as extra arguments (essentially an α-reduction) or by a global variable. This latter is particularly messy and dangerous since first the present value must be saved, then the new value assigned before use. On exit the old value must be restored. This reveals another problem, *viz* error exits: it is still likely to be desirable to restore the previous value of the global in this case. This means that all possible return paths must be covered by an ERRORSET

followed by an assignment of the saved value to the global and then the propagation of ERROR. Alternatively this may be hidden in a form called UNWIND-PROTECT.

In the calculus, an idealised machine would replace a free variable reference with a reference to the expression which would yield its value. As a consequence of the reduction process (be it normal or applicative order), sooner or later the expression should be replaced by its value and thus the free reference is satisfied. Work by [Wadsworth 71] has shown that either technology or computer architecture (or both) are inadequate to implement λ-reduction directly (i.e. the α, β and η operations). Alternatively this may show the folly of trying to mimic a mathematical concept too closely by machine and perhaps indicates a lack of a deeper understanding of the mechanics of reduction. Whatever the pros and cons of this debate, McCarthy (et al.) did not try to copy the reduction process exactly in the first implementation of LISP. The outcome was the use of dynamic scoping and applicative order evaluation, both of which have received severe criticism over the past few years.

The purpose of dynamic scoping is to provide an on-the-fly β-substitution mechanism, so that a free variable reference is satisfied by finding the value bound to that name, rather than by substituting an expression for the name and waiting for the evaluation process to reduce that. In some measure this explains the adoption of applicative order evaluation despite the attractive characteristics of normal order. The other reason, remarked upon in passing in the history, is what was then known as the FUNARG problem [Moses 70]. Its development is charted in greater detail in the latter part of Chapter 0. Normal order evaluation

could be provided by creating a closure of each actual parameter defining expression and binding that to the formal parameter. Left at this stage the solution corresponds to call by name semantics. Alternatively, on the first interrogation of the parameter, the continuation could be evaluated, and the result *replace* the present binding of the identifier (known as call by need). Whatever the implementation scheme, the point to be made is that dynamic scoping is a fundamental part of forming the link between LISP and λ-calculus and its omission from the language is a serious dilution of semantics and expressive power. Purely lexically scoped LISPs are much weakened languages since the programmer must find ways to effect the β substitutions required by other means as outlined above.

Local and dynamic variables

The foregoing should not obscure the fact that many bound variables are only locally (lexically) scoped in practice, that is they have only been introduced to rename an expression (α-reduction). This information can be used advantageously to aid efficiency in compiled LISP, when the bindings of such 'local' variables may be stored and loaded from the function frame, rather than using the more complex procedure associated with variables whose dynamic scope is greater than that of the λ-expression in which they are bound. Indeed, it is more common for variables to be used locally than free. Dynamically scoped variables are important because they permit an elegant method of parameterising large bodies of code; in some sense they are used more to *direct* rather than *contribute* to the computation. It is here that an interesting dichotomy arises; the mechanism used in compiled code to provide free variable lookup is also the backbone of the interpreter (and must be so to permit compiled and interpreted code to be interleaved) -

except that the interpreter uses the dynamic mechanism to bind *all* variables. This means (with some notable exceptions [Blair 78] [White 82]) that all variables are treated as variables which might be free elsewhere. The interpeter does this by default (because the classical binding methods cannot take this into account) whilst the compiler must be told whether a variable use is local or free and generates code accordingly. This leave a semantic trap for the unwary, and more importantly denies the definition of a compiler as a semantics preserving program transformer. There are three solutions to this:

(i) treat all variables as if they would be used 'free' when compiling as is the case with INTERLISP [Teitelman *et al.* 72]

(ii) proclaim that the compiler and interpreter have different semantics and that the programmer must beware! This is the case with MACLISP [Moon 78] and most of the other widely used dialects.

(iii) extend the interpreter to get it right

A feature which distinguishes the binding method developed here from the others is the availability of a direct handle on the environment and on the relationships between the different environments. This makes it possible to provide both local and dynamic scoping of identifiers in multiple environments within the interpreter. As remarked earlier the majority of variables are only used locally (i.e. within the λ-expression where they were bound), so it is reasonable to default to local binding and only create it such that the binding is freely visible when expressly requested. For pragmatic purposes this distinction must be known at

binding time because the local and fluid bindings are kept in separate structures. In addition, it must be done this way because the local binding is only a function of the interpreter and must not impinge on the efficiency of compiled code which already supports local variables by its own method, whereas free (fluid) variables are shared between the two parts. There is still one area of uncertainty arising when compiled code calls the interpreter (EVAL). This question is discussed in greater detail in the next section. The mechanism for variable access etc. is largely unchanged, the only cost being in the overall speed of the interpreter, but this must be weighed against the semantic advantages of mixed local/fluid declarations and in particular the very positive benefit of ensuring that all compilation does is make the program run faster, not faster and differently!

Before explaining how the method will work there is one open question which was passed over in the preceding paragraph: how to determine whether a binding is local or free. The existing mechanism in widespread use (and also adopted in Cambridge LISP) is arranged by the functions FLUID and UNFLUID (or SPECIAL and UNSPECIAL depending on the dialect). The semantics of these 'functions' are rather curious. Their purpose is to communicate to the compiler that the variables in the argument list are to be treated as free variables. UNFLUID removes this information. However, these functions cannot occur in the body of a function as that would be compiled as an invocation of the function (UN)FLUID to take place when executing the function in which the call occurs. To acheive the desired effect the invocation must appear at the outermost (i.e. global) level, hence their scope is global or indefinite. So, although their implementation is functional (in the impure sense), their nature is declarative. Yet it should be necessary to consider this

information only within the scope of a single function. Also, this global usage may lead to other occurrences of the same variable unintentionally being bound as fluid which raises two objections:

(i) the program (taken as a whole) may not be as efficient as it could be

(ii) the program may not behave as predicted (and the bug is somewhat more subtle to detect than a straightforward omission of a FLUID declaration)

There seem to be two solutions to this problem:

(i) as used in MACLISP, LM-LISP and COMMON LISP, the DECLARE statement is either embedded in a function or at top level, e.g.

```
(DECLARE (SPECIAL FOO))
```

(ii) describing the mode of a variable in the formal parameter list as in LISP 370, e.g.

```
(LAMBDA ((FLUID FOO)) ...)
```

It is reasonable to default to creating a local binding of a variable; there are three reasons for this:

(i) usage as a local is more prevalent than as a free variable

(ii) the fluid mechanism is more expensive (in compiled code that is) and so it is beneficial to encourage the use of the cheaper methos when the former is unnecessary and, more importantly, unintentional

(iii) it is important that occurences of free bindings should be significant and require positive action on the part of the programmer rather than happen by accident.

Now to describe the binding process itself and the associated variable access procedures. For each variable in the formal parameter list, the first stage is to determine whether it is to be bound local or free - this will, of course, depend on the mode declaration scheme chosen. If the mode is free, the existing binding process is followed (see Chapter 3); conceptually a new environment-value pairing is joined to the front of the value chain of an identifier. If the mode is local a new environment-value pairing is constructed as previously, but it is joined to the front of the local value chain. Needless to say this will provide local scoping in *multiple* environments unlike the method described in [White 82].

Variable interrogation now first searches the local chain using the criterion that the correct value is the one in the pair whose environment is *equal* (nay, even *EQ*!) to the current environment. This implies a search O(n) where n is the number of local bindings of a name. In fact this can be improved by the corollary that should any environment met in the search be an ancestor of the current environment, then there is no local binding of that name and so the free binding must be sought using the existing algorithm. Note that the existing algorithm searches for the

first *equal* or *ancestor* binding whilst local binding is checked by *equal* or *not ancestor.* In some part this shows the power and flexibility of a scheme in which relationships between environments (or binding nodes) are encoded explicitly, rather than using the implicit relations existing in an ordinary data structure. The actions for assignment and unbinding can easily be deduced from their descriptions in the initial implementation in Chapter 3.

A suitable notation for this is still under development as part of a rationalised version of LISP called LIER [Padget & Fitch 84]. As always the hardest part is finding the best compromise between convenience and accuracy. It is attractive to proclaim that *all* variable references will be treated as local and that free variable references must be embedded in a distinguishing form (hence any other free reference would be regarded as an error). This is a means of ensuring that the code is strictly carrying out the programmer's intent. This attitude also possesses a symmetry in that free variables must be distinguished both at the point of declaration (binding) and use. It is not clear whether this imposes an unreasonable burden. A particular consequence is that the CAR of most forms would have to be embedded, since all functions are fluid and this therefore implies a free reference. Alternatively, one admits that the usage of the function position of a form is generally different from the argument positions and waives the embedding requirement.

Outstanding problems

Although this scheme tidies the semantics of LISP it still leaves two problems: one is related to the function SET, which has always created difficulties; the second is merely a deficiency in the underlying systems (i.e. Cambridge LISP and PSL but not YKTLISP).

Firstly, the problem still remains that it is not possible to provide named access to local variables in compiled code without abandoning the use of frame locations to hold local values. Instead exactly the same mechanism as employed in the interpreter must be used, rather than one that appears the same in all but this case. Consider the two following (admittedly pathological) functions:

```
(setq x 2)

;; named access to a local variable

(de foo (x)
  (print x)
  ((lambda (y) (print (eval y))) 'x)
  (print x))

output after (foo 1)  interpreted compiled
                          1          1
                          1          2
                          1          1
then evaluating x         2          2

;; named assignment to a local variable

(de bar (x)
  (print x)
  ((lambda (y) (set y 5) (print x)) 'x)
  (print x))

output after (bar 1)  interpreted compiled
                          1          1
                          5          1
                          5          1
then evaluating x         2          5
```

In the absence of any guidance, the compiler has made 'x' a local variable and the name 'x' has been lost, so that when it comes to be accessed or modified the local 'x' no longer exists by name, and the free binding of x is used/updated, which is in conflict with the behaviour of the intepreted function. Given that it is desirable to continue the practice of compiling local variables and references to them away into frame locations, how can these matters be reconciled? One course of action would be to restrict SET to work only on fluid bindings (i.e. ignoring locals) in the interpreter (a case for the reintroduction of

CSET?). Then the second function in the above would at least be consistent. Unfortunately the problems runs deeper. This solution cannot of course be applied to EVAL, since it is the heart of the interpreter and is used to interrogate both local and fluid bindings, so seeking to control this problem at the variable access/update level is incorrect. It would not be possible to write either of these functions (or produce analogues of the cases they exemplify) without the QUOTE function, or some means of proscribing evaluation (e.g. FEXPRS and MACROS).

Having identified the source of the problem (ie. delayed evaluation) does not lead to a solution unfortunately - apart from the drastic measure of excising such forms from the language. The better strategy is to avoid losing the names of local variables in compiled code by maintaining information about names and respective frame locations - in essence a display. Naturally such a scheme will make compiled application of EVAL (to atoms) and SET more complex, but given the relative infrequency of use in this manner, performance is not likely to be unduly affected, although it does increase the cost of building a frame.

The second matter alluded to above is more ironic than a major cause for concern. A method has been described for implementing mixed local/fluid variable bindings in multiple environments as an extension of the new binding scheme presented herein. The Cambridge LISP system (and indeed PSL) is based around a single control stack (partly a consequence of BCPL), so that although multiple environments using local variables will work interpreted, they will not work compiled! The solution to this is obvious, but involves a lot of work and is presently beyond the scope of this thesis. The necessary ideas for managing

multiple control environments are quite well understood [Bobrow & Wegbreit 73], and could be taken as a basis for future work. Initial ideas suggest that a more thorough investigation of this problem (in conjunction with consideration of garbage collection, in particular stacking CONS) would be beneficial.

Chapter summary

This chapter is something of an oddity; it does not belong directly to the main subject matter of the thesis and yet it is initimately tied up with it both as a pre-cursor (the section on λ calculus and scoping) and as a consequnce (the provision of local variables and general scoping). It also sets out areas for further research into more rigourously formalised dialects of LISP and the practical issues that must be solved in order to implement what could be viewed as a 'complete' system (e.g. multiple control environments and named access to local variables).

CHAPTER 7

Memory Management

Garbage collection

The matter of garbage collection is broken into two areas: first, the classical problem of recycling redundant memory which arises in all list processing systems and second, a question that is particular to this scheme, the recycling of environment labels. Environments are created, used and discarded in the same way as other structures. The recovery of the storage associated with them falls within the compass of the existing garbage collection mechanism just as with the other multiple access environment models. In discarding an environment, an environment label is lost and is also therefore a candidate for recycling. When an environment is discarded part of the tree labelling has also been thrown away, so although the tree is still self-consistent it is not as compact as it might be. For this reason it is worth considering how the evaluation tree might be relabelled. Although the rate of utilisation of labels is fairly low in relation to the limit on the size of a label imposed by the immediate representation of integers, it is reasonable to be concerned about the approach of that limit.

The immediate representation of a LISP integer is an object the same size as a machine address in a tagged architecture (M68000, NS16032, IBM 370 series). This is not true in the case of an untagged architecture (e.g. VAX), where the following observation does not hold. As long as label usage is dense (a property which can be maintained by an extension of garbage collection, see the section on relabelling the environment tree in this chapter), the system must run out of memory

(addressability) before it can run out of labels. The reasoning is that because an environment descriptor, which is the only place an environment label can occur, is a larger structure than an immediate integer and the maximum integer is also the highest address, then memory must be exhausted before the limit on the size of an integer is reached. This lemma can *only* be preserved if there are no gaps in the use of labels. An environment descriptor occupies eight words in the present implementation. An immediate integer is, of course, one word. This suggests that only one eighth of the labels can be used before reaching the top of memory. In practice this ratio will be smaller because other data structures will also be filling up the heap. Altogether this means that there is a large margin for overshoot in the utilisation of labels. It is also worth remarking that because there is more than a factor of two between the usage of labels and the allocation of memory there must be a garbage collection before the labels reach overflow. Therefore if relabelling is an integral part of garbage collection it is impossible for the labels to exceed their limit.

Implications for storage reclamation

Despite the fact that the various LISP systems to which this new binding regime has been added have different garbage collection methods (mark and sweep in Cambridge LISP and stop and copy (in several forms) in PSL) the strategic modifications have been very similar. Over the development of the new scheme two approaches have been taken to effect the physical representation of the model. This experience has served to highlight the importance of the form and connectedness of the data structure (or structures) used to model the environment, and how it affects the cost of garbage collection.

For the initial system (and some long period of its development), the physical representation was closely allied to the naive conceptual model. That is, the environment was a chain of environment descriptors as shown in Figure 3.1 and the value chains were constructed as association lists rooted in the identifier to which they referred. When considering an identifier in garbage collection in the original system, the practice was simply to mark the value cell, but the value cell now contains a more complex object; not all of the entries in the value chain may be accessible if they are bindings from a once preserved and now discarded environment.

Laying aside the question of uninterned identifiers - since that would only complicate the discussion - all the identifiers must be accessible from the object list (oblist). Therefore by traversing this structure all the value chains can be visited, but what are the criteria to decide which elements of the value chain should be marked? If the binding environment of a value is an ancestor of the current environment, then clearly that pair can be marked. That however is not sufficient: the object of the whole exercise is to reintroduce environments as values available for the use of the programmer, therefore some of the values may themselves be environments. So a secondary condition is that if the binding environment itself has been marked, then the pair should be marked. This creates yet another problem: a newly valid environment structure may be found whilst scanning the oblist, which may retrospectively make invalid bindings valid, so to ensure that everything that should be marked is marked, it is necessary to scan the oblist repeatedly until there is a pass in which nothing is marked.

This implies a minimum of two scans and is potentially very

expensive. An alternative but somewhat questionable strategy (which could only work on virtual memory systems) takes advantage of the fact that for housekeeping purposes, the environment descriptor contains a list of the variables bound in that environment. Given this information, the garbage collector could mark the identifiers by recursing - of course that could uncover more environment values and hence more recursion. At this point in the development of the system, environment descriptors were constructed from LISP vectors and were tagged as such which made such a solution much more difficult to integrate because of the non-recursive marking routine. (That is when ascending a structure it is not possible to distinguish between an environment descriptor or a LISP vector - neither can one use a flag to indicate state because there is no 'recursion' to save it in the correct context).

That constitutes the first part of tracing the oblist; marking has been restricted to those environment/value pairs whose environments are still accessible. That means that the value chain of an identifier contains some marked and some unmarked cells in the same structure; to rectify this matter another pass is made over the oblist (and any uninterned identifiers found on the way) applying a function to clean the value chains. This simply descends the chains, splicing out unmarked pairs, leaving a structure in which all the marks are correctly and consistently placed. Now it is safe to proceed with relocation and compaction phases.

As can be seen from the above, garbage collection in a multiple environment system seems rather complicated - but it need not be so. The process in deep binding is straightforward; there is a structure rooted at the current environment. Every live environment can be

reached (and hence traced) from there. There are two reasons why garbage collection got out of hand:

(i) a separate datatype was needed for environment descriptors

(ii) variables and their binding environments were not explicitly linked. There was only an implicit relationship in the existence of an environment/value pairing on the value chain of an identifier. This all points to a need for a better data structure.

These observations lead to the construction of a new physical representation of the environment (as described on Chapter 3) which had the twin advantages of changing unbinding to constant cost per variable and making the whole environment a properly connected structure, so that just as with deep binding by tracing from the current environment, everything which should be marked will be marked. At the same time some new datatypes were introduced into the system: the environment descriptor, the continuation and the binding vector. The continuation and the environment descriptor were included largely for cosmetic reasons, but the other was critical to the garbage collection strategy.

The binding vector contains all the bindings and the linkage information for a particular environment, and internally it has a rather unusual format. The value chain of an identifier, rather than being constructed from CONS cells to make an association list, becomes a doubly linked list, the adjacent cell containing the environment value pair (see Figure 3.2). These four words (two links, environment and value) are part of a vector containing all the information about bindings made in

that environment. The links in fact refer to positions inside other (binding) vectors, but other bindings in the value chain may no longer be accessible (in the strict sense) so it is important to prevent information in such binding vectors from being marked. This is why it is essential to have a separate datatype for binding vectors. Normally all the entries in a LISP vector should be traced and marked; for a binding vector only each fourth entry (the value) need be marked. There is no need to trace the environment pointer since that is how the binding vector was reached in the first place, and the reasons for not touching the up/down links have already been explained. The value chains still need to be cleaned in a manner similar to that described before, and so some indicator must be left to show that a particular binding should be retained in the value chain structure. To this end, the code which traverses a binding vector puts a mark in each of the downward links. The unlinking process is notionally as previously discussed with only some detail changed. Thus overall garbage collection cost (as much as this can be quantified) is a little more than for existing multiple environment systems, because of the need for an unlinking phase.

Although it is not usual to take account of garbage collection time when running system benchmarks, in analysing a technique with such fundamental effects, the question must be considered. A radical improvement in execution speed cannot always recompense for profligate use of space, since this will increase the frequency of garbage collection and could lead to an overall lengthening of the elapsed time for the computation. Both space and time were a particular concern in the initial association list based system, not so much that there was a high turnover of memory, but because the actual cost of garbage collection was at least 2-3 times higher - and had no upper bound. The present

version is much better in this respect being bounded and around 1.5 times as expensive.

This does not address the question of how frequently garbage collection occurs. It is always a problem emulating multiple environments within a system only intended to handle programs with a simple stack-like behaviour. One must perforce use heap to allocate these environments and yet frequently program behaviour will be stack-like, leading to a higher rate of memory turnover where previously the space would have been allocated and reclaimed by using a stack. There are two ways to combat this:

(i) manage function frames in a separate heap [Bobrow & Wegbreit 73]

(ii) optimise allocation and reutilisation of certain structures, that is by maintaining freelists.

Eventually, some technique to support (i) is desirable, but not always practicable, certainly in the short term, so to ameliorate the rate of memory turnover, the second line of reasoning was pursued. Two major sources of memory usage can be identified - environment descriptors and binding vectors. Because of the variable nature of the latter, it was decided that the overhead involved in managing and using a freelist of those objects would probably outweigh any advantage gained for garbage collection. Environment descriptors however are fixed size objects and hence easier to handle. The tactic is very simple: on leaving a contour, attach the environment descriptor (ED) to a freelist, as long as the environment it describes has not been preserved. When

creating a new contour, an ED is taken from the free chain if possible, otherwise new memory is allocated at the same time as for the binding vector. At garbage collection this freelist is discarded so that the space can be recovered by the system.

Relabelling the environment tree

As remarked at the opening of this chapter there is a secondary 'garbage collection' problem peculiar to this model; because the environment label is finite and preserved nodes will certainly be discarded at some time it is important to have a technique for reclaiming the dead nodes and so recycling their labels. In fact it does not strain the analogy to compare the problem (and its solution) with the compacting and relocating collection strategy. The terminal (leaf) nodes of the evaluation tree are identified, then after setting the label generator back to its initial state, the whole tree is traversed and relabelled. Thus all the labels in the tree become compacted (that is, there are no unused labels) and in a sense the environments have been relocated to their new labels, but of course the relationships between the environments have been retained. This is a simplification of the process involved only being intended to draw out the similarity between traditional garbage collection and the relabelling problem.

In practice it seems some restrictions may have to be placed on when it is feasible to relabel consistently. In particular there should only be one 'open' leaf node. That implies that relabelling cannot be done from inside a context switch. A little reflection reveals that this is to be expected and quite reasonable because there is no consistent labelling for a *single* tree which permits either of two nodes to be extended. So providing this criterion is not violated, relabelling may take place as an

integral part of garbage collection.

Why is it even necessary to consider the relabelling problem? Beside the points raised in the opening section of this chapter regarding the limitations on label size there is one other important feature. If two labels are in the same subtree then the ancestry test is O(1) otherwise it is O(n) as discussed in Chapter 2. Relabelling results in a tree in which all the labels are in the same generation subtree and so leads to an overall performance improvement.

Here is an informal description of a simple algorithm to relabel the environment tree which assumes that there is only one currently active leaf node as described above. During the mark phase of the garbage collector all the currently live environment descriptors will be visited. A list of all the terminal nodes (distinguished by the sequence and span of the label being equal) is constructed. This does not utilise any extra store since the nodes can be joined together using the generation entry of the descriptor as a link field because that is now redundant information. This list is ordered by the age of the leaf nodes with oldest first. It is a simple test, achieved by comparing the magnitude of the sequence part of the label. At the end of this stage a thread has been constructed from the left-most to the right-most terminal node of the evaluation tree passing through all the nodes in between in order. The next job is to carry out the actual relabelling. The counters *sequence* and *generation* are set to zero. The method is as follows:

for each terminal node

(i) descend to the root of the evaluation tree counting the

nodes which have not yet been relabelled

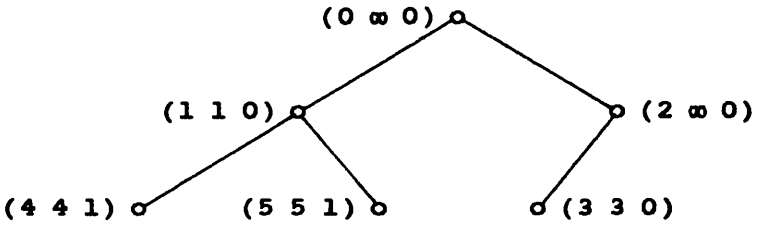
(ii) ascend creating new labels on the way

sequence is taken from the counter *sequence* which is then incremented

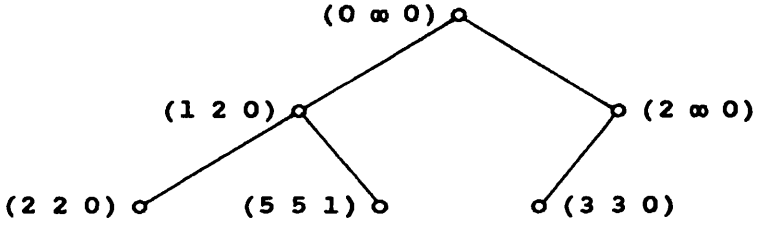
span is set to the value of *sequence* at the base of the tree plus the count of unlabelled nodes calculated in the descent phase

generation is set to zero

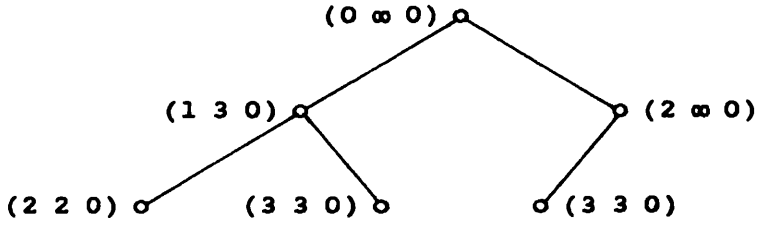
To help in the understanding of this scheme various stages of the relabelling of a tree are shown in Figure 7.0. It now only remains to explain how to determine whether a node has been relabelled or not. The obvious solution is some sort of marker bit but another pass over the tree would be needed to remove them. An alternative method relies on the observation that all environment descriptors have an entry to point at the root of their subtree. In the relabelled tree this entry will have to be the same for all nodes and so by building a new root node (one can ensure that there is enough space left in which to do this) and making the descend and ascend code a little more careful the new root can be used as an indicator. At present there is no facility in ASLISP for relabelling the evaluation tree because the development of a more general algorithm than the above is still sought. The purpose of this section is to show that it can, in principle, be done, albeit with some restrictions.



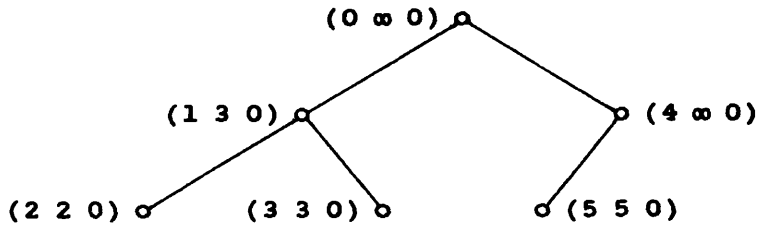
(a)



(b)



(c)



(d)

Figure 7.0

Future work in garbage collection

It is clear from the second section of this chapter that multiple access environments implemented within a single control environment model imposes a not insignificant burden on both interpreter and garbage collector, so it is a matter of continuing research on how this can be improved, if not resolved.

Of prime concern is to find how to take advantage of the frequent stack-like behaviour and allocation of memory [Stallman 80] where it is used to maintain the access environments. This is closely tied up with the allocation of control environments, which are also largely stack-like in behaviour and indeed in many systems are constrained to be so because the control frames are built on a linear structure which does not permit their retention. If the access environments were allocated out of the same linear structure, they can be reclaimed immediately they are no longer required. Of course this is no solution, but only a pointer in the direction of where to look for one, since such a scheme does not admit the preservation of access environments, which was the original purpose of the exercise.

It seems advantageous to pursue the notion of allocating control and access together because this provides a simple mechanism for the recovery, or rather reutilisation, of memory. Once the machinery for such a system exists, it is only a small step from this position to the idea of making CONS create new pairs on top of this 'stack', so that all structures caused by a particular function invocation are associated with that invocation/environment. This has three potential benefits:

(i) locality of reference (of real importance on virtual memory systems)

(ii) retention of a context (and the structure within it) without imposing a garbage collection overhead

(iii) deallocation of a context and the structure within it *en masse*

Points (ii) and (iii) are really two sides of the same coin, and direct consequences of handling storage in larger blocks, rather than at the granularity of the CONS cell. In this model, the physical representation of a context is comprised of four elements:

(i) the environment descriptor

(ii) the control environment

(iii) the access environment

(iv) structure created by the current environment

This is much the same as existing implementations. The key is that (iv) is identifiable as a single object rather than a collection of pointers to various locations in ths store. If the local context can be identified in this way, it seems this may permit the garbage collector to ignore it in some sense, because all the cells may be declared live (at some cost of retaining unreferenced cells within a context). In the same way when leaving or discarding an environment, it may be possible to declare the whole local structure dead and hence reclaim that space.

A problem in both cases is the matter of external references: in the first, references to external structure may be hidden in the local structure, and so were the local context and the external context to be relocated relative to each other, there might be some difficulty in resolving these cross-context links. In the second, references into local structure may be retained by such means as assigning to a free variable or by returning a result. Of course neither of these restrictions applies in the case of assigning or returning an atomic value. The next question is how to detect these intrusions; the simplest place would appear to be at their creation, such as in the functions SET, SETQ, RPLACA and RPLACD using reference counts on the function frame. Non-local values can be recognised by virtue of the memory address, because the addressing itself is organised into segments (corresponding to a function invocation) and pages within these segments, so that as a particular invocation requires more memory it is allocated to and associated with it by address. The idea of reference counts on storage blocks owes something to the garbage collector in Interlisp-D [Bobrow & Clark 79] [Bobrow 80] (descended from the BYTELISP implementation on the Alto [Deutsch 80] [Deutsch & Bobrow 76]) as indeed does the use of local (in page) and non-local (out of page) address. The applications however are somewhat different, their intention being the mangement and collection of self referential list structure using reference counting. More recently a similar approach was suggested in [Liebermann & Hewitt 83] as an extension of [Baker & Hewitt 77] and [Baker 78a], but this seems overly complicated, places far too much responsibility on the microcoding of primitive operations, makes many of the basic LISP functions relatively expensive and above all is merely a *gedanken* exercise.

Chapter summary

The problems encountered during the extension of the original garbage collector have been described. The initial approach was crude and expensive, indeed no upper bound could be placed on the time required to complete the process. A better understanding of the problem lead to a fairly drastic revision of the underlying data structures, which in turn transformed the process into one taking bounded time and only marginally more costly than the original. A side effect was an improvement in the cost of unbinding.

The discussion then turns to the matter of collecting environment labels. Although the question has been considered quite closely and an interim solution developed (but not implemented), this is very much a matter for further research. The (personal) objection to the interim scheme stems from the restrictions as to when it may be applied. Informal ideas suggest that these restrictions can be overcome at the price of a little more complexity. An attractive possibility is the idea of a continuous relabelling algorithm, that is one which is always relabelling the tree in a manner similar to an incremental garbage collector. The major problem with this is maintaining a consistent relationship in the labelling all the time. This may be feasible by using negative and positive labels by analogy with from- and to- space in copying collectors.

CHAPTER 8

Applications

Use of continuation

This chapter is divided into four major sections identifying the areas which the author has investigated so far for algorithms which might benefit from the use of continuations. These are computer algebra, data base query evaluations, tree searching and, at a more abstract level, the implementation of objects (in the Smalltalk [Ingalls 80] and Actor [Hewitt *et al.* 79] sense). It should be stressed that this chapter discusses potential applications of continuations. In all cases a small amount of code has been written and tested (prototyped) to see if the metaphor is suitable for the problem addressed. The programs are purely exploratory and nowhere near approach the needs of production systems.

Algebra

A large part of the material of this section appears in the paper of Appendix 0 [Fitch & Padget 84], but some additional commentary is in order as is a discussion of some extensions of the ideas presented there. The paper describes a medium sized algebra system supporting rational polynomials, which is capable of solving several widely recognised algebraic problems (f and g series [Sconzo *et al.* 65], series reversion [Hall 73] and the Y(2n) problem [Campbell 72]). It became apparent that continuations used to provide lazy evaluation, or more accurately *directed* lazy evaluation (an idea akin to the annotation scheme in [Burton 82b]), might be advantageous in some of the basic algorithms for manipulating polynomials [Padget 82]. This was brought

about by considering the methods of Altran [Brown 73] and in particular the sparse polynomial multiplication mechanism [Johnson 74]. A program description of the algorithm is given in Figure 8.0 below (simple univariate case).

```
% multiply polynomials a and b
(de pl* (a b)
     (cond
          ((numberp a)
           (cond
               ((numberp b) (times a b))
               (t (pnl* b a))))
          ((numberp b) (pnl* a b))
          (t (pl+ (tpl* (term a) b) (pl* (nterm a) b)))))

% multiply a term and a polynomial
(de tpl* (a p)
     (cond
          ((numberp p) (cons (term a) (times (coeff a) p)))
          (t (conz (tl* a (term p)) (tpl* a (nterm p))))))

% add polynomials a and b
(de pl+ (a b)
     (cond
          ((numberp a)
           (cond
               ((numberp b) (plus a b))
               (t (conz (term b) (pl+ (nterm b) a)))))
          ((numberp b) (conz (term a) (pl+ (nterm a) b)))
          ((equal (exp (term a)) (exp (term b)))
           (conz (tl+ (term a) (term b)) (pl+ (nterm a ) (nterm
          b))))
          ((lessp (exp (term a)) (exp (term b)))
           (conz (term b) (pl+ a (nterm b))))
          (t (conz (term a) (pl+ (nterm a) b)))))

% where  term - leading term of polynomial
%        nterm - reductum of polynomial
%        exp - exponent of a term
%        conz - a semi-lazy cons which suspends the second
         argument
```

Figure 8.0

It is useful to be able to compute the terms of the product in order (where the ordering is system dependent, but either highest to lowest degree or vice versa) for two polynomials P and Q:

$$P(x)\ Q(x) = \left[\sum_{i=0}^{n} a_i\ x^i \right] Q(x)$$

where n is the leading degree of P and $a_i x^i$ is the ith term of P. In a dense representation, this objective is realised quite straightforwardly: all the individual products for summing lie along the diagonal of a matrix as in Figure 8.1.

This is not the case, however, for sparsely represented polynomials. The terms do not appear regularly. Consider Figure 8.2 which shows the matrix of products for two such polynomials and is annotated to indicate the order in which the terms should appear. Obviously this is not regular or predictable as in the dense case and will depend on the structure of the individual multiplier/multiplicand. A naive solution would be to generate all the terms of the solution and then to sort and merge them. This is expensive in terms of both speed and space, not least because terms can be produced which will later cancel (a phenomenon known as intermediate expression swell). Certain properties of the computation can be deduced: each row (or column) of the matrix represents the products of a single term and a polynomial. The leading term of

$$a_{i-1}\ x^{c_{i-1}}\ Q(x)$$

must be of lesser degree than that of

$$a_i\ x^{c_i}\ Q(x)$$

and indeed all the terms of a single column are in order of descending degree because they represent the polynomial:

Matrix for results of (dense) - $(x^3 + 0x^2 + x + 1)\ (x^3 + x^2 + 0x + 1)$

	x^3	0	x	1
1	x^3 4	0 5	x 6	1 7
$0x$	0 3	0 4	0 5	0 6
x^2	x^5 2	0 3	x^3 4	x^2 5
x^3	x^6 1	0 2	x^4 3	x^3 4

Figure 8.1

Matrix for results of (sparse) - $(x^{11} + x^6 + x^2 + 1)\ (x^7 + x^4 + x^2 + 1)$

	x^7	x^4	x^2	1
1	x^7 8	x^4 10	x^2 11	1 12
x^2	x^9 6	x^6 9	x^4 10	x^2 11
x^6	x^{13} 3	x^{10} 5	x^8 7	x^6 9
x^{11}	x^{18} 1	x^{15} 2	x^{13} 3	x^{11} 4

Figure 8.2

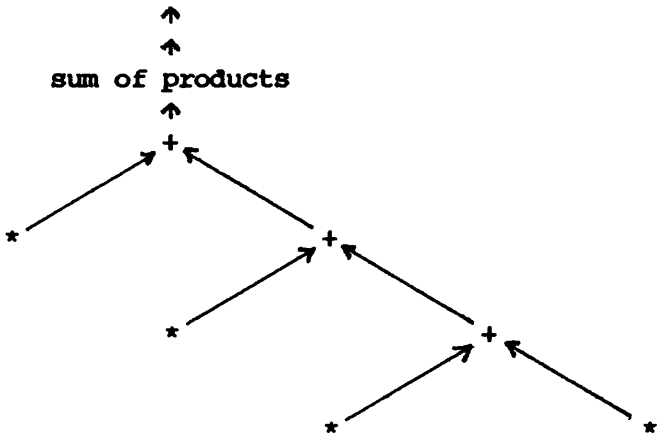


Figure 8.3

$a_j x^{c_j} Q(x)$ where $a_j x^{c_j}$ is the jth term of P(x).

The question is how to merge these polynomials to form the final product in such a way that the terms are generated in order? The program given above constructs a tree of communicating processes arranged as in the diagramof Figure 8.3. The * nodes are simple demand driven generators which produce successive terms of the intermediate polynomials. Control is really invested in the + nodes which arbitrate between the two streams fed to them, returning the higher degree term (and demanding another of that stream) or adding the leading term of each stream if they are of equal degree (and demanding a new term of each). The program also has the intriguing behaviour that a + node will splice itself out of the tree (or more likely remove itself from the head of the tree since those generators should be exhausted earlier) when either of its generators are exhausted. This considerably complicates any formal cost analysis such that it is only possible to determine an upper bound which in practice would never be reached. The methods described in [Johnson 74] use two schemes for organising the intermediate products, a heap (so that probe = O(n log n)) and a list (probe = O(n)). The program given here is conceptually equivalent to the latter. Interestingly Johnson reports that for all but the most wildly unstructured polynomials, list insertion performs better in practice than the heap because of the extra overheads involved.

A similar technique can be applied to division, but a rather curious dilemma arises because division returns two results: a quotient anda remainder. Therefore the answer cannot be represented as a single stream because, firstly, it would be difficult to distinguish the remainder part when reached and secondly, many algebraic algorithms examine the

remainder before using the quotient. The primitives quotient and remainder are quite easy to supply; it is the transition between the two that creates the problem. A naive definition of divide might be

```
(de pi/ (a b) (cons (pquo a b) (prem a b)))
```

but this is extremely wasteful because calculating the remainder necessitates the calculation of the quotient. The way round this seems to be to generate all the quotient (i.e. eagerly), then leave the remainder lazy as soon as it is discovered, and return the pairing of one completely evaluated object and one partially evaluated object.

More recently further features not documented in the paper have been added to the system. These additions all have one thing in common: the use of continuatiosn to provide an object-like mechanism (includinga means of describing memo-functions). Current research is directed at a more comprehensive use of objects throughout the system based on the ideas of [Norton & Marks 84], see also the last section of this chapter.

One of the new facilities introduced supports the description, generation and manipulation of infinite power series. Once defined (by some set of initial conditions), the series may be treated like an object, the messages sent to it being requests for the n^{th} approximation. In this first implementation, the specification of the power series follows that outlined in [Harrington 78] and shares some concepts with the system built on top of Scratchpad [Griesmer & Jenks 72] described in [Norman 75] in that (as indeed must be the case for infinite series) termsare only calculated when needed. In the simplest cases a series could be

defined by naming an expansion variable and giving a coefficient function. for example

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

where the expansion variable is x and the coefficent function is 1/i! This presumes that all series start at zero which is not the case and so a normalisation function is required to translate the origin in effect e.g.

$$\frac{e^x}{x^2} = \sum_{i=0}^{\infty} \frac{x^{i-2}}{i!}$$

where the normalisation function is $1/x^2$. The final generalisation concerns the exponents of the terms of the approximation. At present one is constrained to integral exponents increasing by one for each form of the expansion: clearly this is too limiting. The necessary generality can be provided by one more parameter which will be taken as the denominator of the power for each term: consider the power series representation of sin x:

$$\sin x = \sum_{i=0}^{\infty} \frac{(-1)^i\, x^{2i+1}}{(2i+1)!}$$

where the coefficient function is $(-1)^i/(2i+1)!$ and the denominator of the power is i/2i+1.

Thus a power series is defined by four parameters:

(i) the expansion variable

(ii) the coefficient function

(iii) the normalisation function

(iv) the denominator of the power

The answer to a message sent to a series object is a polynomial representing the approximation taken up to the nth term, where n was the message. This work needs to be extended to consider actual operations on the series themselves (such as addition, multiplication, exponentiation and inversion), rather than the minimal expansion facility available at present.

Another mathematical object which is difficult to handle because of its unbounded nature (and has therefore received little attention in the symbolic algebra field) is the continued fraction. Continued fractions have long history, the first description being attributed to Lord Brouncker, the first president of the Royal Society, in 1624. A continued fraction expansion can provide a rational approximation to fractions, roots and analytic functions amongst others. Such expressions can be written for example

$$1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}}$$

which also conveys how they can be computed, but conventionally they are written so:

$$\frac{1}{1+}\ \frac{1}{1+}\ \frac{1}{1+}\ \frac{1}{1+}\ \cdots$$

This continued fraction approximates the Golden Ratio $\frac{(\sqrt{5} - 1)}{2}$

For a given continued fraction expansion

$$b_0 + \frac{a_1}{b_1 + \frac{a_2}{b_2 + \frac{a_3}{b_3 + \ \ddots}}}$$

which may also be written as

$$b_0 + \mathop{K}_{n=1}^{\infty} (a_n/b_n)$$

the term $\frac{a_n}{b_n}$ is the nth *partial approximant*. The nth approximant can be computed by the following difference equations:

initially $A_{-1} = 1, \quad A_0 = b_0, \quad B_{-1} = 0, \quad B_0 = 1$

and $$A_n = b_n A_{n-1} + a_n A_{n-2}$$

$$B_n = b_n B_{n-1} + a_n B_{n-2}$$

Thus it can be seen that the state of an expression is captured in the variables A_{n-1}, A_{n-2} and B_{n-1}, B_{n-2}, and further approximants may be computed assuming the existence of a function to compute a_n, b_n pairs. The strong relationship between the mechanism to support power series and the mechanism to support continued fractions is now more obvious. As an example, consider the code in Figure 8.4 which constructs an expansion to generate approximations to square roots of integers (this code is not optimal in the sense that it is known that continued fraction representations of square roots are periodic). Also note that the numerator of the partial approximant for the square root

approximation is always unity. hence the optimisation in the computation of the recurrence relation.

```
(de cfsqrt (n)
     (prog (a_n b_n r_n irootn A_n-1 B_n-1 A_n-2 B_n-2)
          % initialise the generators
          (setq a_n (setq irootn (isqrt n)))
          (setq r_n (difference n (times an an)))
          (setq b_n (quotient (plus irootn a_n) r_n))
          (setq A_n-2 1)
          (setq B_n-2 0)
          (setq A_n-1 irootn)
          (setq B_n-1 1)
          % value is a function which returns successively better
          % approximations to √n
          (return (closure
               (prog (A_n B_n)
                    % compute and roll the CF recurrence relation
                    (setq A_n (plus (times b_n A_n-1) A_n-2))
                    (setq B_n (plus (times b_n B_n-1) B_n-2))
                    (setq A_n-2 A_n-1) (setq A_n-1 A_n)
                    (setq B_n-2 B_n-1) (setq B_n-1 B_n)
                    % then crank the partial approximant generator
                    (setq a_n (difference (times b_n r_n) a_n))
                    (setq r_n
                      (quotient (difference (times a_n a_n)) r_n))
                    (setq b_n (quotient (plus irootn a_n) r_n))
                    % the latest answer
                    (return (cons A_n B_n)))))))
```

Figure 8.4

Gauss employed continued fractions in extensions of the work by Euler. Lambert and Lagrange in studying hypergeometric series and ratios of such series. For example

$$\arctan z = \cfrac{z}{1 + \cfrac{z^2}{3 + \cfrac{4z^2}{5 + \cfrac{9z^2}{7 + \ddots}}}}$$

Obviously the mechanism for computing the nth approximant (either symbolically or numerically) remains unchanged from that used in the square root code. Only the function to compute the partial approximants is different. which is as might be expected. Representing such analytic functions by symbolic rational approximation provides a similar service to that of the power series. It is known [Jones & Thron 80] (p.202) that

for numerical approximation continued fractions are very accurate and fast because of very high rates of convergence. However, some questions hang over their use in symbolic approximation: this is largely due to problems in computing the error bound when truncating and the behaviour at poles and branches, particularly in the case of composition of approximations.

A more general method of approximating power series was later developed by Frobenius and Pade, resulting in the Pade table, the entries of which are known as Pade approximants. These are also closely related to continued fractions but as yet no work has been done on their implementation using the above techniques (see [Geddes 79] for other computational methods). It has been suggested [Czapor & Geddes 84] that Pade approximants will be of use in the solution of the differential equations which arise during the course of the Risch integration algorithm [Risch 69].

Database queries

This subject has been the focus of much research over the last ten years, especially the relational model propounded by E.F.Codd. A particular example of the scheme with a novel control structure mechanism is CODD (COroutine Driven Database) described in [King & Moody 83] and in detail in [King 79]. The coroutine implementation follows that described in [Moody & Richards 80], whose operations have recently been reiterated in [Haynes & Friedman 84].

A query in the command language of the DBMS is broken into several smaller simpler steps, then in the compilation and subsequent execution of the query each of these steps maps to an individual

coroutine. The full chain of coroutines corresponds to the complete query. This can be best explained by an example. Consider the (informal) request:

List all books	by Jane Austen	published after Pride & Predjudice
1	2	3
main clause	qualifier	qualifier

The query can be broken into three distinct parts: a main clause and two qualifiers as shown. To expose the nature of the solution consider those three distinct parts as a generator and two filters: now the structure becomes clearer:

```
DB  =>  books  =>  Jane Austen  =>  after P & P  =>  list
          (d)          (c)             (b)          (a)
```

The main clause is rather like a filter as well since it ensures that the tuple ejected from the database are book descriptors, but its operation is somewhat more complex than the other two stages of the pipeline and so the distinction should be preserved. It can now be seen that there is a strong similarity between the physical implementation of the query and a lazy evaluating stream. Given a demand to print an answer, stage (a) requests a tuple from (b) which requests one from (c) which requests one from (d) which extracts one from the database. (c) checks that the tuple satisfies the condition that the author field contains Jane Austen, if so it is passed back to (b). if it fails the test another tuple is demanded of (d) until the list is exhausted or a tuple passes the test. A similar process takes place when (b) receives a tuple: thus any tuple reaching the listing process must satisfy all the conditions in the original query and so is displayed or kept to make a

new relation.

The advantage of employing such an evaluation mechanism is

(i) the simplicity and (perceived) elegance of the processing of queries

(ii) very few intermediate results exist

(iii) (a corollary of (ii)) efficient usage of main memory (no large intermediate data areas) and efficient usage of disk space (large intermediate results might have to be written to disk).

The latter two points are not so significant in the first example but are very important in the case of projections, joins and computing transitive closures of relations [King 79].

The control mechanism in the query described above is quite simple and falls naturally into the style of LISP using recursive function calling. This is significant when the query structure is tree-like, but this limitation to a tree structure can severely affect performance in practical applications. It is observed in [King 79] that the 'compiled' queries contained common subexpressions or, say, in computing two relations for keeping, one was the complement of the other. Both of these cases could give rise to a large amount of duplicate processing. By introducing the *copy* node and the two output *join* node considerable improvements in efficiency were made. Unfortunately this changes the structure of the query from a tree to a directed acyclic graph which precludes the use of recursive function calling to process it. Instead a

genuine coroutine mechanism is needed.

This can be provided by the primitive CONTINUEAS. which given a continuation transfers the return address of the caller to the callee and switches to the context specified in the continuation and evaluates the body. This can be viewed as a combination of GOTO and a context switch which brings us back to the question of full jumps discussed in [Strachey & Wadsworth 74] and in Chapter 1. Until ASLISP is augmented with multiple control environments this operator cannot be supported.

Two other powerful features described in [King 79] are the *collate* and *build* operations. These also serve to confound the desire for a tree structured query: *collate* takes input from several places in the pipeline or pipelines. *build* (which acts like a daemon) is used in the compilation of the transitive closure of a relation and is activated by a tuple reaching the *build* node. It is a disadvantage of the pipeline philosophy that the tuples passing through must be in key order, because this means that in constructing the transitive closure of a relation it is not permissable to make a cycle in the query in order to pass those tuples which have not yet satisfied the relation through the testing mechanism again. A solution to this is found in *build*: it constructs a copy of a specified section of the pipeline above its position in the pipeline (and puts another *build* node at the head) and then lets the computation proceed. Again these features cannot be implemented simply via recursive function call (that is expression continuations) but demand an operation such as was described in the previous paragraph.

Tree searching

This technique is at the heart of most artificial intelligence and expert

system programs (although many applications arise outside this field, AI is probably the most extensive user).

Knowledge bases are generaly organised hierarchically, that is as n-ary trees. Without loss of generality this discussion can ignore questions such as the contents of the nodes of the tree and how the links within the tree are represented. Using such a knowledge base often involves searching from one position until information which fits the requesting template is found. It may be that this "first fit" is not deemed sufficiently good by the procedure that initiated the request and the "next fit" is required. Thus if the search is programmed in such a way that it can suspend itself when a solution is found, when it is subsequently resumed it is easy to find the next solution. This technique is often called backtracking search.

As an example of how continuations might be used in this kind of application a small program is presented in Figure 8.5 which demonstrates a suspending tree walk, in this case as part of a solution to the well known question of whether two binary trees have the same set of leaves (in order) i.e. the same fringe problem. The function *leaves* acts as a kind of generator on the tree structure passed to it as argument, remembering by means of a continuation every occasion on which it is forced to make a decision between two alternative paths. The decision procedure in this case is trivial since the program is designed to make an in order traversal of the tree, but the principle remains the same, and could be extended as outlined above. The other function *seqeq* is also generally applicable: it is simply intended to compare the contents of two streams for equality, so any two structures which can be linearised using a generator can be compared with this function.

```
(de seqeq (s1 s2)
     (cond
          ((null (cdr s1))
           (cond
               ((null (cdr s2)) (eq (car s1) (car s2)))
               (t nil)))
          ((eq (car s1) (car s2))
           (seqeq (sdr s1) (sdr s2)))
          (t nil)))

(de leaves (tree cont)
     (cond
          ((atom tree) (cons tree cont))
          (t (leaves
               (left tree)
               (closure (leaves (right tree) cont))))))
```

Figure 8.5

A more concrete application of continuations to control backtracking search arises in augmented transition network (ATN) parsers. Such a parser has been sketched, based on the description given in [Charniak *et al.* 80]. The program itself is relatively small, but that is as large as it is ever likely to be since that implements the core of the ATN interpreter. The breadth of the parser's capabilities is determined by the number and complexity of grammar rules defined. This means the system could be extended and enhanced just by adding more rules because the whole process is data driven. It is hoped to pursue this topic more actively in the near future.

The object based model of programming

A model of programming and program structuring which has recently come to the fore is known as object oriented programming. It has the advantage of enforcing the kind of structuring into programs and systems which previously depended on the morals and standards of the programmer. Another consideration is the extra security provided, in particular for non-primitive data types, where these are implemented as objects themselves.

There are three reasons which make the object abstraction valuable:

(i) structural integrity

(ii) modular integrity

(iii) resource integrity

These will be discussed in turn, following a brief description of the FLAVOR system [Weinreb & Moon 81] and alternative implementation schemes. Structural integrity is exemplified by the use of an object to provide an abstract data type (in the example below, a stack). Because the support mechanism and data structure of the stack are hidden it cannot be inadvertently or intentionally abused - it is secure. Modular integrity relates to how complex facilities (e.g. editors, compilers) which are comprised of many functions can be viewed as objects with just a few well defined entry points (methods) (e.g. editf, editv, editp or compile, comprop) whilst the rest of the machinery remains inaccessible to the user. Resource integrity refers to how an object can be used to protect resources (e.g. disks, files, network nodes, etc in an operating system). This bears a strong resemblance to structural integrity, but is sufficiently different still to demand a separate heading. In the same way that an object can be used to constrain operations on a data structure it can also constrain the operations on a resource [Hewitt & Atkinson 79].

The programmer who avails himself of such an abstract data type need not concern himself with any details of the physical representation (indeed it would be possible to change it without affecting the user's program as long as the responses to external stimuli were the same).

This highlights how the modularity inherent in the concept is advantageous; this modularity also ensures that there is no chance whatsoever of the user interfering, either maliciously or accidentally, with the functions and variables used to implement the datatype. This last sentence starts to reveal the connection between the ideas of objects and their provision using continuations, since a continuation involves environment capture; why should that environment not capture the set of values which describes an instance of some higher level data structure? Then, to use a particular instance of an object (which is a closure), it is applied to some argument (often referred to as message-passing) which is interpreted by the object, some operation is executed, and a result returned. This view of objects using closures is not new, indeed it has been considered in several systems (e.g. [Weinreb & Moon 81]) and rejected on grounds of efficiency.

The FLAVOR system in LISP m/c LISP is somewhat static in nature, and in particular the relationship between objects is set in concrete once they are compiled. This last restriction is a consequence of the compilation and message decoding schemes used; when compiling a FLAVOR (in particular the message dispatch table), all the messages which may be received by the FLAVORs this FLAVOR inherits are collected to form a single jump table. Thus, when handling a message, this table is used like a *switchon, case* statement or a computed goto. This is the heart of the problem; once a FLAVOR (or any of its ancestors) has been so compiled, the definition of new methods will not have any effect on it. What is needed is a more general dispatching mechanism but of similar efficiency. The tags of the *switchon* used in the FLAVOR system are atoms, and so it might be thought that this would demand a linear search (=> O(n) dispatch cost). However it is generally possible to

establish some ordering on a set of atoms. This ordering could be alphanumeric, but it is also reasonable to use relative position in store, although a linearising compacting garbage collector is necessary to preserve the latter predicate. Given such an ordering the jump table may be scanned using binary search (=> O(log n) dispatch cost). It cannot be expected to exceed this bound, but it can at least be equalled. The existing scheme is limited because of the linear data structure, but fast because it is sorted and can be viewed as a tree.

Can flexibility be found without sacrificing speed? Obviously a more flexible structure is an explicit tree, but that in itself cannot guarantee the necessary performance, since that is only obtained when the branching factor is regular, so the extra sophistication of the AVL-tree is adopted. Recent results [Gonnet 83] suggest that there may be better methods than the AVL scheme for balancing trees; this will be investigated at a later date. The cost of adding a new method to a class may now be higher since the tree must be rebalanced but such an operation is infrequent relative to the invocation of a method. Using the AVL-tree, the time taken to decode a message is O(log n). So in summary: this implementation of objects instantiates an object by means of a closure, so the nature of the object is much closer to the rest of the system, and uses AVL-trees to monitor the list of methods understood by a class.

The use of a tree for storing the methods allows a class to be enhanced at any time whether it is compiled or interpreted, but when taken in conjunction with environment labelling an interesting new potential casn be seen; specialisation or refinement of a class. The key to this is apparent in the way in which environment labelling provides

a very powerful and simple handle on environments and the relationships betwen them; the label can be used to scope arbitrary data structures as long as the accessor functions know how to use that information. Thus the method entries in the tree may be labelled according to the context in which they were created; because refinements will only be made in the context of an existing instance, the methods will be added to the tree and labelled with that context. The only instances able to 'see' those methods will be that which was originally refined and any further instances descended from it.

A provisional syntax has been developed and some examples of its use are given in Figure 8.6 which defines a class called stack.

A class is defined as follows

```
(defclass  <name-of-class>
           <instance-var-list>
           <class-properties>
           <method-definitions>)
```

<name-of-class> is self explanatory

<instance-var-list> is of the form

```
((<instance-var-1> <default-value-1>)
 ...
 (<instance-var-n> <default-value-n>))
```

where <instance-var-i> is a variable name and <default-value-i>is its default initial value at the creation of an instance. This may be overridden at creation time if the class has that property.

<class-properties> are specified as a list of atoms:

(i) get: generate methods to access the instance variables

(ii) set: generate methods to modify the instance variables

(iii) new: instance variables may have new values assigned at creation time

(iv) augment: methods may be added to this class (that is there will be a handler for the addmethod message)

(v) excise: methods may be removed from this class (that is there will be a handler for the remmethod message)

(vi) redefine: methods may be redefined (redefmethod message)

<inheritance-list> a list of classes from whom to inherit methods

<method-definitions> a list of methods.

This last argument is optional. The messages (and their associated methods) are defined at the declaration point of the class. This is particularly important for highly secure classes which would normally preclude the addition of methods. A similar effect could be achieved by removing the addmethod and remmethod facilities once the extra definitions are complete.

```
(defclass
    stack                          %name of class
    (s '(stackl-bottom))           %one instance variable and initial value
    ()                             %no extra properties (for security)
    ()                             %nothing to inherit
    ((empty
       (lambda (s) (eq (car s) 'stackl-bottom)))
     (push
       (lambda (s i)
         (rplacd s (cons (car s) (cdr s)))
         (rplaca s i)))
     (pop
       (lambda (s)
         (cond
           ((eq (car s) 'stackl-bottom)
            (error ...))
           (t (prog1 (rplaca s (cadr s)) (rplacd s (cddr s)))))))
))
```

Figure 8.6

Therefore the class stack has a single instance variable whose default value is the list (stackl-bottom). It has no other properties so its integrity cannot be violated. It is a basic structure and so has nothing to inherit. Finally the methods for the three known messages are defined (in addition to DESCRIBE and CLASS which are generated automatically). These are defined at the point of declaration of the class since it was decided not to permit augmentation of the class for security reasons. Thus an invocation of (stack) returns an object which is an instance of a stack, and it is initially empty. The only possible operations on this object are DESCRIBE, CLASS, EMPTY, PUSH and POP; it can be shown informally that those do not undermine the consistency of the data struncture and thus we have structural integrity. Initial investigations show that this mechanism lends itself well to the description of algebraic data structures such as polynomials, and handles well the question of polynomials over different fields and factorisation domains. In this respect the scheme appears to share some features with SCRATCHPAD [Jenks 84], though that is an operator centred system whilst this is type centred. A considerable amount of work in this direction has latterly

been published in reports on the NEWSPEAK system [Foderaro 83].

What all this serves to show is that objects provide a way of typing structures. Although this involves restricting operations on structures, their use does not seem to be as obstructive to the programmer as the strong lexical, and more recently, polymorphic scheme proposed in other languages.

Large pieces of system software such as the LISP compiler are made up of many separate functions, very few of which have any application outside the needs (or context) of the compiler itself. Generally there are a few well defined entry points to this morass of code e.g. *compile* and *comprop*. The existence of all those functions raises two concerns:

(i) modification or redefinition of internal functions (intentional or otherwise)

(ii) the large proportion of the namespace occupied by functions irrelevant to the user

A crude solution to this position is to REMOB all but the functions to which the user should have access. This is not very attractive and again raises the debate over interned and uninterned identifiers. One attempt to resolve the matter is the concept of the block compiler in which the set of functions comprising a module are compiled together with a defined set of entry points. This causes all references to the internal functions to be 'compiled away', an idea analogous to what often happens to non-fluid variables in LISP compilation, so such functions do not then exist as LISP objects. The drawback with both of these methods

is the removal of the name - which is in some sense what is wanted, since it is what is perceived as the only way of controlling access to the function - because that precludes tracing and debugging of the module. The parallel starts to become clearer; the compiler is an object which understands a few messages (compile, comprop), the majority of its bulk is to be hidden so that it cannot interfere or be interfered with by users programs. That mass can be obscured by ensuring those functions are defined only in the context of the compiler, thus modular integrity is obtained. This does not solve the second question, namely the density of the symbol space, which with the above, is still very much a topic for further research. Some strategies worth investigating are the scoping of the oblist structure, and the use of local oblists (i.e. within objects).

The third case in which objects are useful is in resource protection. Consider the question of controlling access to a file or database. Several programs may safely have concurrent read access but only one writer (and no readers) may have access at one time (i.e. the standard readers/writers problem). Applications programs view the resource as an object to which to send read or write requests (messages). All the queuing, fairness and starvation are dealt with inside the object (informal proofs are possible [Hewitt & Atkinson 76]) and in fact the resource is never 'open' to the application program as is the case in other synchronisation and protection methods (e.g monitors, critical regions, path expressions). It is this that makes it possible to prove that starvation and lock-out are impossible if the implementation of the operations within the object satisfy the aforementioned conditions. That is not such a weak condition as it sounds, indeed it is considerably stronger than any of the other schemes can allow, because inherent in their design is the granting of actual control over the resource for the

desired period. Therefore it cannot be guaranteed that a pathological process cannot usurp the resource. Hence objects can offer a high degree of resource integrity.

Chapter summary

Four application areas for continuations have been discussed: algebra, databases, tree searching and the implementation of objects. There are many more. There has not been time as yet for a detailed investigation of the use of continuations, even in the domains outlined, but the initial response is promising. In the near future it is hoped to spend time on using the results of the more fundamental research that is the basis of this thesis. Continuations provide a very powerful and general means of control although one must beware of trying to apply them inappropriately.

Conclusion

This thesis has identified and explored the feasibility of the only untried method of managing multiple environments, to wit, the technique of associating environment with bindings. Previously the binding was buried in an environment structure (deep binding) or else the run-time system kept track of the changes which took place in the environment in order to be able to undo them at a later stage (full shallow binding or rerooting).

This work is intended to try and solve a practical problem and so to measure how well this aim is achieved several benchmark programs have been run on the new system and on equivalent systems with alternative binding models. The benchmark programs must consider two execution styles: stack behaviour (i.e. simply using function call and return for control and environment) and context switching (where the environment plays an active part in the computation). The results of the former may be regarded as more significant because they are larger, more general and widely used. They indicate that the new model is slower (in the range 4%–15% for application programs) than shallow bound (both without and with rerooting). This is to be expected. The new model is faster than deep binding (and its variants), although the function cell trick sometimes approaches quite closely. That is not significant though, because the technique is available to any scheme and is therefore applicable to the new model if one is prepared to confuse the semantics.

The continuation tests comprise a few simple programs by the author (although the possibilities list program is based on an example in the

INTERLISP manual). Deep binding with the function cell trick often came out best here, but the new model was always second or even beat deep binding with function cells. Rerooting was very bad in one of these tests being even slower than deep binding. It may yet be too early on the basis of these results to form a firm conclusion but it would seem that the initial intention of providing the context switching ability of deep binding with the variable lookup speed of shallow binding has, to a large measure, been achieved.

As the work presented here has progressed, it has become more apparent how important and how fundamental the continuation is both as a programming technique for complex control regimes and for reasoning about and understanding all control mechanisms. Continuations exist in all programs and systems. The problem is that they are almost always implicit (return address on a stack or adjustment of a stack or frame pointer). In many respects this is an advantage: if the program is straightforward in respect of control flow and evaluation context, it would be tedious to write such things explicitly. Occasionally, however, it is vital to have access to this information and to vouchsafe complete responsibility to the programmer. Here this has been done by explicit continuations. This permits deceptively and surprisingly small programs to be written to achieve operations which would either be extremely complex and obscure or even impossible without this facility. The tree searching program is a good example.

The continuation need not be the province of a few esoteric languages (such as those based on combinators and graph reduction), nor is it necessary to discard to dynamic scoping in order to make the closure problem simpler.

The generality of the labelling model leads to facilities which no other scheme can provide (e.g. scoped property lists etc.) which are attractive features for AI programming. The system as it stands would benefit from further work both to improve efficiency and to generalise the implementation. Nevertheless it is conclusive evidence that the environment labelling method is a competitive binding method, combining the advantages of deep and shallow binding with little of their detractions.

References

[Abrahams 66]
P W Abrahams
The LISP 2 Programming Language and System
Proceedings 1966 Fall Joint Conf. pp661-676
AFIPS Press. NJ

[Aho *et al.* 76]
A E Aho. J E Hopcroft & J D Ullman
On Finding Lowest Common Ancestors in Trees
SIAM J. Comp 1976. Vol 5. No 1. pp115-132

[Baker 78a]
H G Baker
List Processing in Real Time on a Serial Computer
CACM April 1978. Vol 21. No 4. pp280-294

[Baker 78b]
H G Baker
Shallow Binding in LISP 1.5
CACM July 1978. Vol 21. No 7. pp565-569

[Baker & Hewitt 77]
H G Baker & C Hewitt
The Incremental Garbage Collection of Processes
SIGPLAN Notices. August 1977. Vol 12. No 8. pp55-59

[Barendregt 81]
H P Barendregt
The Lambda Calculus. its Syntax and Semantics
in Studies in Logic and The Foundations of Mathematics series
North Holland. New York. 1983

[Barron *et al.* 63]
D W Barron. J N Buxton. D F Hartley & C Strachey
The Main Features of CPL
The Computer Journal. Vol 6. July 1963. pp134-143

[Barwise & Perry 83]
J Barwise & J G Perry
Situations and Attitudes
MIT Press. Cambridge. Mass. 1983

[Blair 78]
F W Blair
The definition of LISP 1.8 + 0.3i
IBM Thomas J Watson Research Centre Internal Report

[Blair 79]
F W Blair
LISP/370 Concepts and Facilities
IBM Research Report RC 7771 (#33639). 1979

[Bobrow & Murphy 67]
D G Bobrow & D L Murphy
The Structure of a LISP System Using Two Level Storage
CACM March 1967. Vol 10. No 3.. pp155-159

[Bobrow 80]
D G Bobrow
Managing Reentrant Structures using Reference Counts
TOPLAS July 1980, Vol 2, No 3, pp269-273

[Bobrow & Clark 79]
D G Bobrow & D W Clark
Compact Encodings of List Structure
TOPLAS October 1979, Vol 1, No 2, pp266-286

[Bobrow & Wegbreit 73]
D G Bobrow & B Wegbreit
A Stack Model and Implementation of Multiple Environments
CACM October 1973, Vol 16, No 10 pp591-603

[Boyer & Moore 79]
R S Boyer & J Strother Moore
A Computational Logic
Academic Press, New York, in ACM Monograph series

[Brown 73]
W S Brown
The ALTRAN User's Manual
Bell Laboratories, 1973

[Burton 82a]
F W Burton
A Linear Space Translation of Functional Programs to Turner Combinators
University of East Anglia, Technical Report, 1982

[Burton 82b]
F W Burton
Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs
University of East Anglia, Technical Report, 1982

[Campbell 72]
J A Campbell
SIGSAM Problem #2: The Y2n Problem
SIGSAM Bulletin 1972, Vol 6, pp8-9

[Charniak *et al.* 80]
E Charniak, J Riesbeck & J McDermott
Artificial Intelligence Programming
Lawrence Earlbaum Associates, New York, 1980

[Church 40]
A Church
The Calculi of Lambda Conversion
Annals of Mathematical Studies 6, Princeton University Press, 1941

[Clarke *et al.* 80]
T J W Clarke, P Gladstone, C MacLean & A C Norman
SKIM - The S, K, I Reduction Machine
Proceedings 1980 LISP Conference, pp128-135

[Curry 30]
H B Curry
Grundlagen der kombinatorischen Logik
Amer. J. Math. 1930. Vol 52 pp509-536, 789-834

[Curry & Feys 58]
H B Curry & R Feys
Combinatory Logic Vol I.
North Holland, Amsterdam, 1958

[Czapor & Geddes 84]
S R Czapor & K O Geddes
A Comparison of Algorithms for the Symbolic Computation
of Pade Approximants
Proceedings of EUROSAM 84. Springer Verlag (LNCS) 172

[Deutsch 80]
L P Deutsch
BYTELISP and its ALTO Implementation
Proceedings 1980 LISP Conference. pp231-242

[Deutsch & Bobrow 76]
L P Deutsch & D G Bobrow
An Efficient, Incremental, Automatic Garbage Collector
CACM September 1976. Vol 19. No 9. pp522-526

[Evans 68]
A Evans Jr.
PAL - A Language Designed for Teaching Programming Linguistics
Proceedings ACM National Conf. 1968. pp395-403

[Fitch & Norman 77]
J P Fitch & A C Norman
Implementing LISP in a High-level Language
Software Practice and Experience 1977. Vol 7. pp713-725

[Fitch 82]
J P Fitch
Private Communication

[Fitch & Padget 84]
J P Fitch & J A Padget
A Pure and Really Simple Initial Functional Algebraic Language
Proceedings of EUROSAM 84. Springer Verlag (LNCS) 712

[Foderaro 83]
J Foderaro
The Design of a Language for Algebraic Computation Systems
PhD. Thesis, University of California (Berkeley), 1983

[Friedman & Wise 76]
D P Friedman & D S Wise
CONS should not evaluate its arguments
Proceedings 3rd International Colloquim on Automata, Languages and
Programming. 1976. pp257-284

[Galley & Pfister 75]
S Galley & G Pfister
The MDL Programming Language
MIT LCS Technical Report, 1975

[Geddes 79]
K O Geddes
Symbolic Computation of Pade Approximants
ACM TOMS June 1979, Vol 5, No 2, pp218-233

[Gonnet 83]
G H Gonnet
Balancing Binary Trees by Internal Path Reduction
CACM December 1983, Vol 26, No 12, pp1074-1081

[Griesmer & Jenks 72]
J H Griesmer & R D Jenks
The Scratchpad System
IBM Research Report RC 3925 (#17792), 1972

[Griss & Hearn 81]
M L Griss & A C Hearn
A Portable LISP Compiler
Software Practice and Experience 1981, Vol 11, pp541-605

[Griss *et al.* 82]
M L Griss, E Benson & G Q Maguire
PSL: A Portable LISP System
Proceedings of 1982 ACM Symposium on LISP and
Functional Programming, pp88-97, ACM, New York

[Hall 73]
A Hall
Solution to SIGSAM Problem #3
SIGSAM Bulletin 1973, Vol 26, pp15-23

[Harel & Tarjan 84]
D Harel & R A Tarjan
Fast Algorithms for Finding Nearest Common Ancestors
SIAM J. Comp May 1984, Vol 13, No 2, pp338-355

[Harrington 78]
S Harrington
Methods for Generalized Infinite Power Series
Utah Technical Report UUCS-78-107

[Haynes *et al.* 84]
C T Haynes, D P Friedman & M Wand
Continuations and Coroutines: An Exercise in Meta-Programming
Proceedings of 1984 ACM Symposium on LISP and
Functional Programming, ACM, New York

[Haynes & Friedman 84]
C T Haynes & D P Friedman
Engines Build Process Abstractions
Proceedings of 1984 ACM Symposium on LISP and
Functional Programming, ACM, New York

[Hearn 74]
A C Hearn
A Mode Analyzing Algebraic Simplification Program
Proceedings ACM 74, pp722-724, ACM, New York

[Hearn 83]
A C Hearn
The REDUCE-3 Manual
Rand Corporation, Santa Monica, USA

[Henderson & Morris 76]
P Henderson & J H Morris
A Lazy Evaluator
Proceedings 3rd ACM Symposium on The Principles of Programming Languages, 1976, pp95-103

[Hewitt 72]
C Hewitt
Description and Theoretical Analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot
MIT AI Technical Report 258, 1972

[Hewitt & Atkinson 79]
C Hewitt & R E Atkinson
Specification and Proof Techniques for Serializers
IEEE Transactions on Software Engineering January 1979, Vol 5, No 1, pp10-23

[Hewitt *et al.* 79]
C Hewitt, G Attardi & H Lieberman
Specifying and Proving Properties of Guardians for Distributed Systems
in Semantics of Concurrent Computation, Springer Verlag (LNCS) 70

[Ingalls 78]
D H H Ingalls
The Smalltalk-76 Programming System: Design and Implementation
Proceedings 5th Symposium on Principles of Programming Languages, pp9-16, ACM, New York

[Johnson 74]
S C Johnson
Sparse Polynomial Arithmetic
Proceedings EUROSAM 74, SIGSAM Bulletin 1974, Vol 8, pp63-71

[Jones & Thron 80]
W B Jones & W J Thron
Continued Fractions, Analytic Theory and Application
Vol 11 of The Encyclopedia of Mathematics and its Applications, Addison Wesley

[Kennaway 82]
J R Kennaway
The Complexity of a Translation of λ-calculus to Combinators
University of East Anglia Technical Report 1982

[King 79]
T J King
The Design of a Relational Database Management System
PhD. Thesis. University of Cambridge. 1979

[King & Moody 83]
T J King & J K M Moody
The Design and Implementation of CODD
Software Practice and Experience 1983. Vol 13. pp67-78

[Lampson & Pier 80]
B W Lampson & K A Pier
A Processor for a High-performance Personal Computer
Xerox Corporation. PARC Technical Report. 1980

[Liebermann & Hewitt 83]
H Liebermann & C Hewitt
A Real-time Garbage Collector based on the Lifetimes of Objects
CACM June 1983. Vol 26. No 6. pp419-429

[McCarthy 60]
J McCarthy
Recursive Functions of Symbolic Expressions and their Computation
by Machine - Part I
CACM April 1960. Vol 3. No 4. pp184-195

[McCarthy *et al.* 62]
J McCarthy. P W Abrahams. D J Edwards. T P Hart. & M E Levin
The LISP 1.5 Programmer's Manual
MIT Press. 1962

[McCarthy 63]
J McCarthy
A Basis for a Mathematical Theory of Computation
In Computer Programming and Formal Systems.
(eds. P Braffort & D Hirschberg). pp33-70
North-Holland. Amsterdam

[McCarthy 77]
J McCarthy
The History of LISP
Proceedings Conf. on The History of Programming Languages
ACM. New York. 1977

[Maier 79]
D Maier
An Efficient Method for Storing Ancestor Information in Trees
SIAM J. Comp 1979. Vol 8. No 4. pp599-618

[Marti *et al.* 79]
J B Marti. A C Hearn. M L Griss. C Griss
The Standard LISP Report
SIGPLAN Notices. Vol 14. No10. pp48-68

[Moody & Richards 80]
J K M Moody & M Richards
A Coroutine Mechanism for BCPL
Software Practice and Experience. 1980. Vol 10. pp765-771

[Moon 78]
D Moon
The MACLISP Manual
MIT, Cambridge, Mass.

[Moses 70]
J Moses
The Function of FUNCTION
SIGSAM Bulletin 1970, Vol 4, pp13-27

[Norman 75]
A C Norman
Computing with Formal Power Series
ACM TOMS 1975, Vol 1, No 4, pp346-356

[Norton & Marks 84]
A Norton & J O Marks
Private Communication

[Padget 82]
J A Padget
Escaping from Intermediate Expression Swell: a Continuing Saga
Proceedings of EUROCAM 82, Springer Verlag (LNCS) 144

[Padget 83]
J A Padget
The Ecology of LISP, or
The Case for the Preservation of the Environment
Proceedings of EUROCAL 83, Springer Verlag (LNCS) 164

[Padget 84]
J A Padget
A New Binding Model for PSL
Utah Technical Report

[Padget & Fitch 84]
J A Padget & J P Fitch
The Rationale of LIER: a considered LISP
Proceedings of RSYMSAC 84, RIKEN, Tokyo

[Richards 69]
M Richards
BCPL - A Tool for Compiler Writing and System Programming
Proceedings of Spring Joint Conf. 1969, pp557-566

[Risch 69]
R H Risch
The Problem of Integration in Finite Terms
Transactions American Mathematical Soc. 1969, Vol 139, pp167-189

[Rosser 35]
J B Rosser
A Mathematical Logic without Variables
Annals of Mathematics 1935, Vol 36, pp127-150

[Schonfinkel 24]
M Schonfinkel
Uber Die Bausteine Der Mathematischen Logik
Mathematische Annalen, Vol 92, 1924, pp305-316

[Sconzo *et al.* 65]
P Sconzo, A le Shack & R Tobey
Symbolic Computation of f and g series by Computer
Astronomical Journal 1965, Vol 70, pp269-271

[Scott & Strachey 71]
D Scott & C Strachey
Toward a Mathematical Semantics for Computer Languages
Technical Monograph PRG-6, Oxford University, 1971

[Sleator & Tarjan 83]
D D Sleator & R E Tarjan
A Data Structure for Dynamic Trees
Journal of Computer and System Sciences 1983, Vol 26, pp362-391

[Stallman 80]
R M Stallman
Phantom Stacks
MIT AI-Memo 556, 1980

[Sussman & Steele 75]
G J Sussman & G L Steele
SCHEME: An Interpreter for Extended Lambda Calculus
MIT AI Memo 349, 1975

[Stoye *et al.* 84]
W Stoye, T J W Clarke & A C Norman
Some Practical Methods for Rapid Combinator Reduction
Proceedings of 1984 ACM Symposium on LISP and
Functional Programming, ACM, New York

[Strachey & Wadsworth 74]
C Strachey & C P Wadsworth
Continuations, A Mathematical Semantics for handling full jumps
Technical Monograph PRG-11, Oxford University, 1974

[Sussman & McDermott 72a]
G J Sussman & D V McDermott
The CONNIVER reference manual
MIT AI Memo 259, 1972

[Sussman & McDermott 72b]
G J Sussman & D V McDermott
From PLANNER to CONNIVER - A genetic approach
Proceedings 1972 Fall Joint Conf. pp1171-1179
AFIPS Press, NJ

[Teitelman *et al.* 72]
W Teitelman, D G Bobrow, A K Hartley & D L Murphy
BBN-LISP TENEX Reference Manual
Bolt, Beranek and Newman, August 1972

[Teitleman 78]
W Teitelman
The INTERLISP manual
Xerox Corporation, 1978

[Turner 79a]
D A Turner
A New Implementation Technique for Applicative Languages
SWPE Vol 9, 1979, pp31-49

[Turner 79b]
D A Turner
Another Algorithm for Bracket Abstraction
The Journal of Symbolic Logic, Vol 44, No 2, pp267-270, 1979

[Wadsworth 71]
C P Wadsworth
Semantics and Pragmatics of the Lambda Calculus
DPhil Thesis, Oxford University, 1971

[Weinreb & Moon 81]
D Weinreb & D Moon
The LISP Machine Manual
Symbolics Corporation, 1981

[Weizenbaum 68]
J Weizenbaum
The FUNARG Problem explained
MIT, Cambridge, Mass, 1968

[White 82]
J L White
Constant Time Interpretation for Shallow Bound Variables
in the Presence of Mixed SPECIAL/LOCAL Declarations
Proceedings of 1982 ACM Symposium on LISP and
Functional Programming, pp196-200, ACM, New York.

Appendix 0

A Pure and Really Simple Initial Functional Algebraic Language

J P Fitch & J A Padget

published in Springer Verlag (LNCS) 174

A Pure and Really Simple Initial Functional Algebraic Language

J. P. Fitch & J. A. Padget.
School of Mathematics.
University of Bath.
Claverton Down
Bath, England.

Abstract

A medium sized algebra system supporting rational functions and some elementary functions, which is written in the purely functional subset of LISP is described. This is used to investigate the practicability of writing systems in a no-side effect, no property list, pure style. In addition, using the experimental LISP system in Bath that allows for full environment closures, ways have been discovered in which eager (applicative) evaluation and lazy (normal) evaluation strategies can be applied to computer algebra. The system is demonstrated on some well known sample programs.

Introduction

Since the early days of computer algebra, systems have been written in LISP. However in general, they have employed the extended version of LISP that is known as LISP 1.6 [Quam & Diffie68] and its descendants. One feature of all these programs is their use of side effects with both global and fluid variables, and the object-oriented use of the property list. In this way the programming language used has become divorced from the mathematical model of lambda calculus which bore it. More recently, and especially after the Turing lecture by Backus [Backus78], there has been a revival of interest in the pure functional, zero assignment and single assignment languages. Evidence of this is the rise of projects such as SKIM [Clarke et al. 80], ALICE [Darlington & Reeve81], AMPS [Keller et al. 79] on the hardware side, engendered by the work of [Turner79] and [Burton & Sleep82]. A particular reason for this interest is that in this programming style new architectural concepts of reduction machines and parallelism are immediately applicable. This is an alternative to the approach of [Marti & Fitch83].

An open question which has hung over the future of these elegant schemes is whether it is practical to write large systems whilst still remaining within the constraints imposed by functional purity. Viewed from a mathematical standpoint there is no doubt that it is feasible, if only by writing a Turing machine simulator, but the concern of this paper is with the pragmatics of such programs. We wish to discover the practical problems in writing a system with the functional paradigm, both in the resulting efficiency of the code, and the intellectual effort required on our part.

In writing such a demonstration system, the authors had a choice of base

language. By building on the ASLISP dialect of LISP (a compatible extension of Cambridge LISP [Fitch & Norman77]) [Padget83] [Padget84], it is possible to delay the decision of whether to use normal order or applicative order evaluation. ASLISP is an experimental system that provides an efficient implementation of full environment closures by a method of environment labelling [Padget & Fitch]. With this new tool we can experiment with mixed eager and lazy evaluation (by explicit closures) in the same program. This is equivalent to the node labelling techniques of Burton [Burton82] in a practical context. Another benefit of the availability of closures is that high order functions can be applied in a sophisticated manner to overcome the self imposed discipline of programming style to provide an elegant solution to problems which do not lend themselves easily to the functional metaphor.

Throughout this paper we have used a MC68000 based computer running the Tripos operating system, both for our new system and for the implementation of REDUCE we use for comparisons [Fitch83].

In order to test the system and compare various evaluation strategies, the now old set of test programs, the f and g series [Sconzo et al. 65], the Y(2n) problem [Campbell72] and the series reversion problem [Hall73] have been coded and run.

System Design and Implementation

In a previous paper Fitch and Marti [Fitch & Marti82] described NLARGE, a small algebra system for use on a microcomputer which manipulates rational forms based on multivariate polynomials. As described in that paper, NLARGE is written in a functional style but not completely pure. It uses a polynomial representation to contruct rationals which it makes canonical by always dividing out the greatest common divisor, and ensuring that the denominator is positive. This system was taken as a starting point for the new functional system and a large number of modifications were made to remove all assignments and the destructive use of the property list of atoms. This involved extensive use of embedded lambda expressions to give the effect of assign-once variables, the passing of functions as arguments and of course a heavy reliance on the compiler for the removal of tail recursions. For the majority of the functions of NLARGE this modification was straightforward. The main areas of difficulty arose in the parsing of the input language, and a section below is devoted to this part of the system. Apart from this, the form used for looping constructs was rather contorted and was hard to follow. As an example we present in figure 1 the function for raising a polynomial to an integer power.

The basic data structure used for polynomials is the same as that in NLARGE, that is the REDUCE variant of the recursive data structure, but extended to allow elementary functions as kernels. This common data structure is obviously well suited for a functional programming style, which to a large extent can be seen in the current REDUCE sources.

The fundamental algorithms of the algebra system are addition, multiplication, subtraction and division. A simple implementation of all these can follow the NLARGE code except where there is a need for the calculation of a gcd. This is the first place in our system where we consider a non trivial algorithm. As the system handles rational forms in canonical representation the gcd algorithm is fundamental to the system. This function in NLARGE was the furthest from the required pure style, and so the opportunity was taken to improve the algorithm used, the reduced PRS algorithm in NLARGE, to the subresultant algorithm in Parsifal (as the new system is called). It is with pleasure that we can report that the functional implementation of the subresultant algorithm is shorter in code than the procedural reduced polynomial remainder sequence (as would be expected), but also took less time to write, and considerably less time to debug.

In this main algebraic part of Parsifal we encountered the first problem. When running interpretively all was well, but when we attempted to compile the system some deficiencies in the compiler were noted. The compiler we use is a descendent of the Portable Lisp compiler [Griss & Hearn81], which deals well with lambda expressions in the function position of a form, and compiles a separate function for lambda expressions as arguments. There are circumstances when the compiler should be forced to declare some variables FLUID, but for good local reasons does not notice. In fact the code shown in figure 1 is such a case; consider the status of the variable fn. This indicates the need for a return to the local functions of LABEL, or some variant of this.

It is apparent that for efficient use of space we are going to need the compiler to be smart about tail recursions. In the simple cases there is no difficulty, but when fluid variables are involved the compiler seems to be over cautious.

In order to make a true algebra system there are a number of other algebraic functions that are needed. So far the only one of these that we have implemented is substitution, which is fairly straightforward, apart from the minor confusion introduced by substituting a rational form into a polynomial.

Parsing and Printing in a Functional Style

Initially it was expected that this would be one of the most difficult tasks, since the use of READ admits side-effects. Of course, sooner or later in any applicative system some form of I/O must be done. It is a question of how well the functional part is insulated from this (i.e. the degree of integration of I/O at the 'implementation' level, or the degree of abstraction viewed at the functional level), and how much of the implementation may be written in a functional style. The obvious model is the stream; however this requires lazy evaluation (explicit coroutining is unacceptable), and hence can only be considered in the second stage system. In the first instance, the programming style is voluntarily limited to being first order functional. This restriction

leads to a compromise between purity and expediency.

The solution chosen also serves as some explanation of the remark above regarding insulation/integration. The top-level system driver reads in tokens until a delimiter (';') is encountered. These tokens are constructed into a list and this is handed over to the parser. Hence the parser itself never has to read, and so manages to remain side-effect free. The parser is a straightforward recursive descent method, only complicated slightly by the need to 'read' tokens from the list which is passed down as an argument to each level, and returned as part of the result with the requisite tokens nibbled off. In this way, the non-functional reading process is kept as far removed from the main body of the code as possible.

The system to which we are moving makes extensive use of the closure facility in ASLISP. This is to great advantage in the parsing process. Being able to 'demand' a new token of the input stream permits a more natural style of coding, although it is still necessary to bind the closure at each level or return the continuation as part of the result to ensure that the correct suspension is evaluated. It is altogther more satisfactory that reading is now even further removed from the body of the system; being hidden inside a stream generator.

The general approach to printing has been similar to reading, where at one level the printer generates a stream of characters, which are printed separately. However we have noticed that as we moved to a lazy evaluation system that in order to preserve a natural print style it seems necessary to evaluate the answer in full before it can be formatted.

Results of Initial System

The system we have so far described is capable of running the f and g series, and SIGSAM examples 2 and 3. For these we present in figures 2, 3 and 4 the user level programs, and in table 1 the timing results, and comparisons with REDUCE. While being considerably slower than REDUCE for the recursive function style, for larger iterative programs it performs credibly. These results are preliminary, as we have not yet attempted any extensive optimization of the system. We expect to make some gains from improved algorithms, but will sustain some loss as Parsifal becomes more general. We have determined that in the present implementation a large overhead results from the macro expansion, for example of for loops and blocks, during evaluation.

Use of Normal Order Reduction

One of the advantages claimed for the pure applicative programming style is that one can use normal order reduction, that is, lazy evaluation. The work of Turner on KRCL [Turner80] makes a major point of the freedom of algorithm that lazy evaluation allows, and the perceived performance of SKIM-1 [Clarke et al. 80] is a clear indicator that we should consider whether the system would benefit from judicious use of normal

order evaluation. In a previous paper Padget [Padget82] indicated that the use of closures gives access to improved algorithms. Such an algorithm is polynomial multiplication. Despite other algorithms with asymptotically good performance, the best practical multiplication algorithm is Johnson's algorithm [Johnson74]. The basic principle of this algorithm is to delay the production of the terms of the product until there is reason to believe that the term may contribute to the answer chain at the end. Described in this way it is clear that Johnson's algorithm is a suitable starting point for the inclusion of some lazy evaluation. The implementation of this algorithm using closures is given in Appendix 1. In Table 2 comparisons are given for some of our simple problems using the lazy Johnson algorithm and the more normal algorithm. At present the timings are a little disappointing, but this may well be due in part to our inexperience in programming with explicit closure, and in part to poor compilation of context switching. In addition since only the multiplication phase has been coded lazy, there is a fair overhead in conversion between the two forms and very few of the advantages of the method have a chance to become apparent.

There are a number of other places in the system where laziness can be usefully applied. We have already mentioned the parser, and we can see other sections of code where we intend to experiment. Division presents an interesting dilemma; the divide function is expected to return a quotient and a remainder, but when evaluted lazily, the remainder would only appear after all the terms of the quotient have been consumed. It is often the case that an algorithm calls for one expression to divide another exactly (done by checking that the remainder is zero), and then make use of that quotient, which will by then have all been evaluated, thus it must be reconverted into the lazy form. Quotient and remainder by themsleves create no particular problems.

Extended Functional Programming

The pure functional style advocated in this paper is of course limited to the programmer. When the functions are compiled we can expect for the time being that the usual von Neumann machine is being used, and the code will involve assignment to registers and goto instructions. In an analogous way we can contemplate an extended pure style in which we allow certain object style functions to exist as an aid to efficiency without affecting the overall purity. Indeed the outlawing of side effects makes one of the main extended forms possible. We refer to memo functions.

If whenever a function is evaluated in a environment the result is remembered, for example on an association list connected to the function, it is possible to interrogate this memory before evaluating the function body to see if the value in this environment has been calculated already. It is well known that the use of a memo function can modify the expected computational time in a non-trivial way: for example consider the Fibonacci numbers by the naive program or the f and g series where we will be able to convert the recursive times to the iterative ones.

Conclusions

This paper has presented an experimental pure functional algebra system written in a dialect of LISP that supports functional closure. While there are many experiments outstanding we have already seen that once one has learnt the style it is possible to write reasonably efficient programs in a fairly short time. The use of some normal order reduction gives us a wider means of expression that we have not yet fully exploited. The system is of medium size, amounting to 20 pages of LISP, and so we cannot yet answer the question on the practicability of writing large programs, although we have noticed a marked shortening of the function based code. To write a REDUCE replacement, for example, would take considerably more time and intellectual effort, but we feel that we have learnt lessons that make us hopeful that such a task is not impossible.

Among the plans we have for continuing to develop Parsifal are to make a fully lazy version, and to implement it under Miranda, Turner's most recent version of his combinator-based language. We have given some thought to the problems introduced by a pattern matching capability, and forsee this as an exciting area for research.

We wish to acknowledge our debt to Dr J B Marti for allowing us such free access to the latest version of NLARGE, and Dr A C Norman who first raised the question of the practicability of the functional style.

Figures

```
(De P^ (a n)
  (Cond
    ((MinusP n) (P^ (Pl/ a) (Minus n)))
    (t ((Lambda (fn) (fn (PCreate 1) 0))
        (Lambda (aa i)
          (Cond
            ((Eq i n) aa)
            (t (fn (P* aa a) (Add1 i)))))))))
```

Figure 1: Raising a polynomial to a power

```
U : - 3 * mu * sig;
V : eps - 2 * sig^2;
W : - eps * (mu + 2*eps);
DbyDt(x) : U*(x DF mu) + V*(x DF sig) + W*(x DF eps);
f(n) : If (n=0, 1, DbyDt(f(n-1)) - mu*g(n-1));
g(n) : If (n=0, 0, DbyDt(g(n-1)) + f(n-1));
f(12);
End;
```

Figure 2: Program for the f and g series (recursively)

```
v[0] : 1;
g[0] : 1;
for(m, 1, 4,
   << v[m] : Sigma(
                Sigma(f[k-s,s] * a^s * c^(k-s)
                      * Sub(gg, b*s+2*(k-s), g[m-k]),
                      s, 0, k),
                   k, 1, m),
      g[m] : Sigma(((gg+1)*k-m)*v[k]*g[m-k], k, 1, m)/m,
      ans[m] : Sub(gg, -2*b, g[m]) >> );
ans[4];
end;
```

Figure 3: Program for SIGSAM Problem 2

```
diff(a, n) :
  sum(e[i]*(a DF e[i-1]), i, 1, n);

wfac(a, b, c, d) :
  if(a=b,
     if(b=c,
        if(c=d, 1, 4),
        if(c=d, 6, 12)),
     if(b=c,
        if(c=d, 4, 12),
        if(c=d, 12, 24)));

y2[0] : 1;
y2[1] : e[0]/2;
sum2[1] : 0;
for(n, 2, 4, <<
    sum2[n] : Sigma(y2[a]*y2[n-a], a, 1, n-1)/2,
    sum4[n] : Sigma(
                 Sigma(
                    Sigma(if((n-b-c-d)<0,0,
                             if(b<(n-b-c-d),0,
                                -wfac(n-b-c-d,b,c,d) * y2[n-b-c-d]
                                 * y2[b] * y2[c] * y2[d]/2 )),
                          b, 0, c),
                       c, 1, d),
                    d, 1, n-1),
    y2[n] : sum2[n] + sum4[n] + e[0]
            * (sum2[n-1]+y2[n-1]) - diff(diff(y2[n-1], n), n)/4
            - diff(diff(sum2[n-1], n), n)/4 + (5/8)
            * Sigma(diff(y2[a], n)*diff(y2[n-1-a], n), a, 1, n-2)
>> );
end;
```

Figure 4: Program for SIGSAM Problem 3

```
y2[1] : (2*e[0]^3 + 6*e[2]*e[0] + 5*e[1]^2)/32

y2[2] : ( - 5*e[0]^4 - 30*e[2]*e[0]^2 + ( - 50*e[1]^2 - 4*e[4])*e[0]
        - 28*e[3]*e[1] - 19*e[2]^2)/128

y2[3] : (14*e[0]^5 + 140*e[2]*e[0]^3 + (350*e[1]^2 + 40*e[4])*e[0]^2
        + (392*e[3]*e[1] + 266*e[2]^2)*e[0] + 442*e[2]*e[1]^2
        + 36*e[5]*e[1] + 96*e[4]*e[2] + 69*e[3]^2)/512

y2[4] : ( - 42*e[0]^6 - 630*e[2]*e[0]^4 + ( - 2100*e[1]^2
        - 280*e[4])*e[0]^3 + ( - 3528*e[3]*e[1] - 2394*e[2]^2
        - 32*e[6])*e[0]^2 + ( - 7956*e[2]*e[1]^2 - 720*e[5]*e[1]
        - 1784*e[4]*e[2] - 1242*e[3]^2)*e[0] - 1105*e[1]^4
        - 1488*e[4]*e[1]^2 - 5564*e[3]*e[2]*e[1] - 1262*e[2]^3
        - 168*e[6]*e[2] - 366*e[5]*e[3] - 234*e[4]^2)/2048
```

Figure 5: Output for SIGSAM Problem 3

		Parsifal	REDUCE
f and g			
(Recursive)	8	58.72	46.66
	12	972.88	757.46
(Iterative)	8	13.18	14.98
	12	81.92	97.50
	18	576.82	730.26
Y(2n)			
	6	6.50	6.64
	8	14.32	9.84
	10	29.22	15.86
Series Reversion			
(Recursive)	4	202.00	89.52
(Iterative)	4	55.06	46.92

Table 1: Timing Results

References

[Backus78]
J Backus
Can Programming be liberated from the von Neumann style?
Comm ACM 21 p613-41

[Burton82]
F W Burton
Annotations to Control Parallelism and Reduction Order Control in the distributed evaluation of Functional Programs
(Preprint)

[Burton & Sleep82]
F W Burton and M R Sleep
Executing Functional Programs on a Virtual Tree of Processors
Proc. Conf. Functional Programming Languages and Computer Architecture

[Campbell72]
J A Campbell
SISGAM Problem #2: The Y2n Problem
SIGSAM Bulletin 22 p8-9

[Clarke et al. 80]
T J W Clarke, P J S Gladstone, C D McLean and A C Norman
SKIM - The S, K, I Reduction Machine
Proc. 1980 LISP Conference p128-135

[Darlington & Reeve81]
J Darlington and M Reeve
ALICE: A multiprocessor reduction machine for the parallel evaluation of applicative languages
Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, 1981

[Fitch & Marti82]
J P Fitch and J B Marti
NLARGEing a Z80 Microprocessor
Proc. Eurocam 82, Lecture Notes in Computer Science 144 p249-55

[Fitch83]
J P Fitch
Implementing REDUCE on a Microcomputer
Proc. Eurocal 83, Lecture Notes in Computer Science 162 p128-136

[Fitch & Norman77]
J P Fitch and A C Norman
Implementing LISP in a high level language
Software - Practice and Experience, 7 p713-25

[Johnson74]
S C Johnson
Sparse Polynomial Arithmetic
Proc. EUROSAM 74, SIGSAM Bulletin 8 p63-71

[Keller et al. 79]
R M Keller, G Lindstrom and S Patil
A Loosely-Coupled Applicable Multi-Procesor System
NCC AFIP V 48 p613-22

[Hall73]
A Hall
Solution to SIGSAM Problem #3
SIGSAM Bulletin 26 p15-23

[Griss & Hearn 81]
A C Hearn and M R Griss
The Portable LISP Compiler
Software - Practice and Experience 11 p541-605

[Marti & Fitch83]
J B Marti & J P Fitch
The Bath Concurrent LISP Machine
Proc. Eurocal 83. Lecture Notes in Computer Science 162 p78-90

[Padget82]
J A Padget
Escaping from Intermediate Expression Swell: A Continuing Saga
Proc. Eurocam 82. Lecture Notes in Computer Science 144 p256-62

[Padget83]
J A Padget
The Ecology of LISP. or
The case for the preservation of the environment
Proc. Eurocal 83. Lecture Notes in Computer Science 162 p91-100

[Padget84]
J A Padget
PhD Thesis. University of Bath (in preparation)

[Padget & Fitch]
J A Padget & J P Fitch
Closurize and Concentrate
(In preparation)

[Quam & Diffie68]
L Quam & W Diffie
Stanford LISP 1.6 Manual
Stanford AI Laboratory Operating Note 28.7

[Sconzo et al. 65]
P Sconzo. A Le Shack and R Tobey
Symbolic Computation of f and g series by Computer
Astronomical Journal 70 pp269-71

[Turner79]
D A Turner
A New Implementation Technique for Applicative Languages
Software - Practice and Experience 9 p31-49

[Turner80]
D A Turner
Recursion Equations as a Programming Language
CREST-ITG Advanced Course on Functional Programming and its Applications.
Cambridge University Press (ed. Darlington. Henderson and Turner) p1-28

Appendix: Lazy Johnson's Algorithm (univariate case)

```
% in the following code some functions require explanation:
%    term - leading term of polynomial
%    nterm - reductum of polynomial
%    exp - exponent of a term
%    conz - cons which suspends the second argument

% add polynomials a and b
(de p!+ (a b)
  (cond
    ((numberp a)
     (cond
       ((numberp b) (plus a b))
       (t (conz (term b) (p!+ (nterm b) a)))))
    ((numberp b) (conz (term a) (p!+ (nterm a) b)))
    ((equal (exp (term a)) (exp (term b)))
     (conz (t!+ (term a) (term b)) (p!+ (nterm a) (nterm b))))
    ((lessp (exp (term a)) (exp (term b)))
     (conz (term b) (p!+ a (nterm b))))
    (t (conz (term a) (p!+ (nterm a) b)))))

% multiply polynomials a and b
(de p!* (a b)
  (cond
    ((numberp a)
     (cond
       ((numberp b) (times a b))
       (t (pn!* b a))))
    ((numberp b) (pn!* a b))
    (t (p!+ (tp!* (term a) b) (p!* (nterm a) b)))))

% multiply term and a polynomial
(de tp!* (a p)
  (cond
    ((numberp p) (cons (term a) (times (coeff a) p)))
    (t (conz (t!* a (term p)) (tp!* a (nterm p))))))
```

		Parsifal	REDUCE
f and g			
(Iterative)	5	12.20	4.34
	8	102.38	20.54
Series Reversion			
(Iterative)	3	206.88	13.88
Y(2n)			
	6	43.56	6.64
	8	118.16	9.84
	10	307.14	15.86

Table 2: Timing Results with Lazy Multiplication Algorithm